编程狂人

Programming Madman

NO.49

推酷

关于推酷

推酷是专注于IT圈的个性化阅读社区。我们利用智能算法, 从海量文章资讯中挖掘出高质量的内容,并通过分析用户 的阅读偏好,准实时推荐给你最感兴趣的内容。我们推荐 的内容包含科技、创业、设计、技术、营销等内容,满足 你日常的专业阅读需要。我们针对IT人还做了个活动频道, 它聚合了IT圈最新最全的线上线下活动,使IT人能更方便地 找到感兴趣的活动信息。

关于周刊

《编程狂人》是献给广大程序员们的技术周刊。我们利用技术挖掘出那些高质量的文章,并通过人工加以筛选出来。每期的周刊一般会在周二的某个时间点发布,敬请关注阅读。

本期为精简版 周刊完整版链接:http://www.tuicool.com/mags/5460cd3ed91b1461c700ac32

欢迎下载推酷客户端体验更多阅读乐趣

目录

前端代码异常监控

作者：raphealguo

前言

我是开发微信图文页一名普通的码农。

近期加班加点上线非常重要的的广告功能：

底部的广告区域有关注公众号的按钮，用户点击之后就会给广告主带来粉丝，给文章所有者带来广告收入。

某天，码农心血来潮，想了解一下每篇文章的图片都来自什么域名，于是加了一段统计脚本....

如此简单的for循环能难得了我，测试啥，直接上线！

投诉来了

下午15:00上完线，下班后突然收到一堆同事电话：我们这边发现广告的关注点不动了，用户好多投诉进来了，看到你15:00上了线，快看看有什么问题！

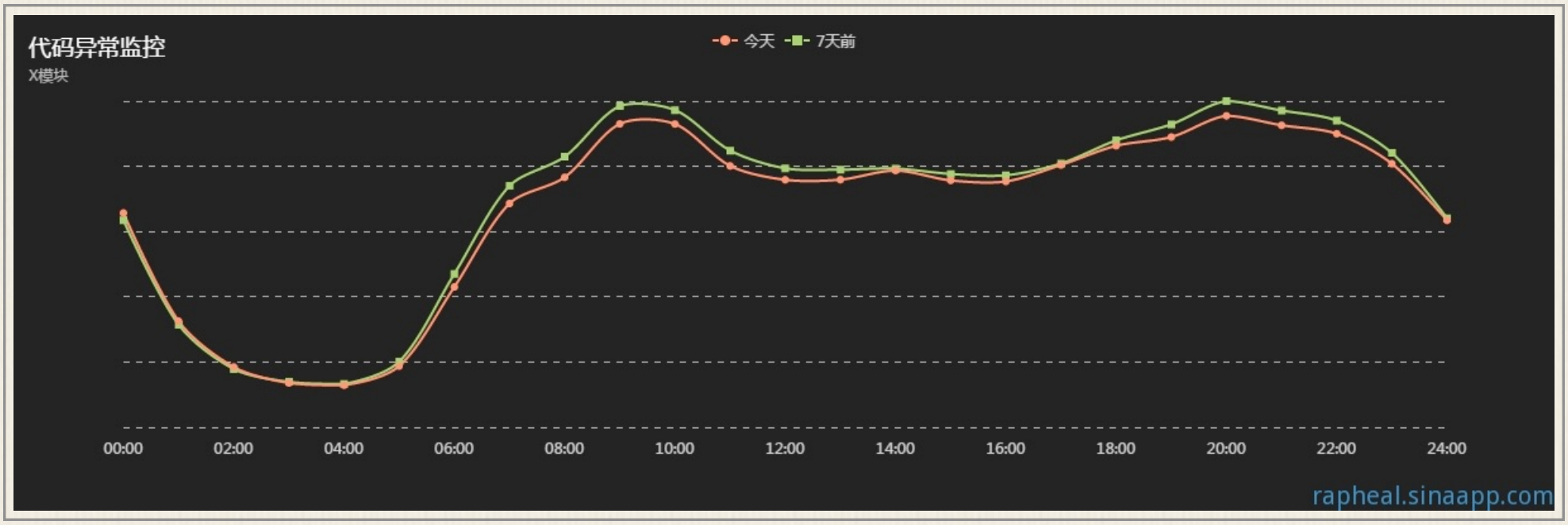
在家VPN简单看了看代码，知道真相后简直无法直视：

把变量len误写成l，导致下边的js脚本不执行了！

反思

对于写代码这件事来说，我们几乎不能避免自己出bug。那我们如果能够在用户侧部署监控，看着用户在我们面前“出错”的话，我们就能很快发现问题并及时处理。

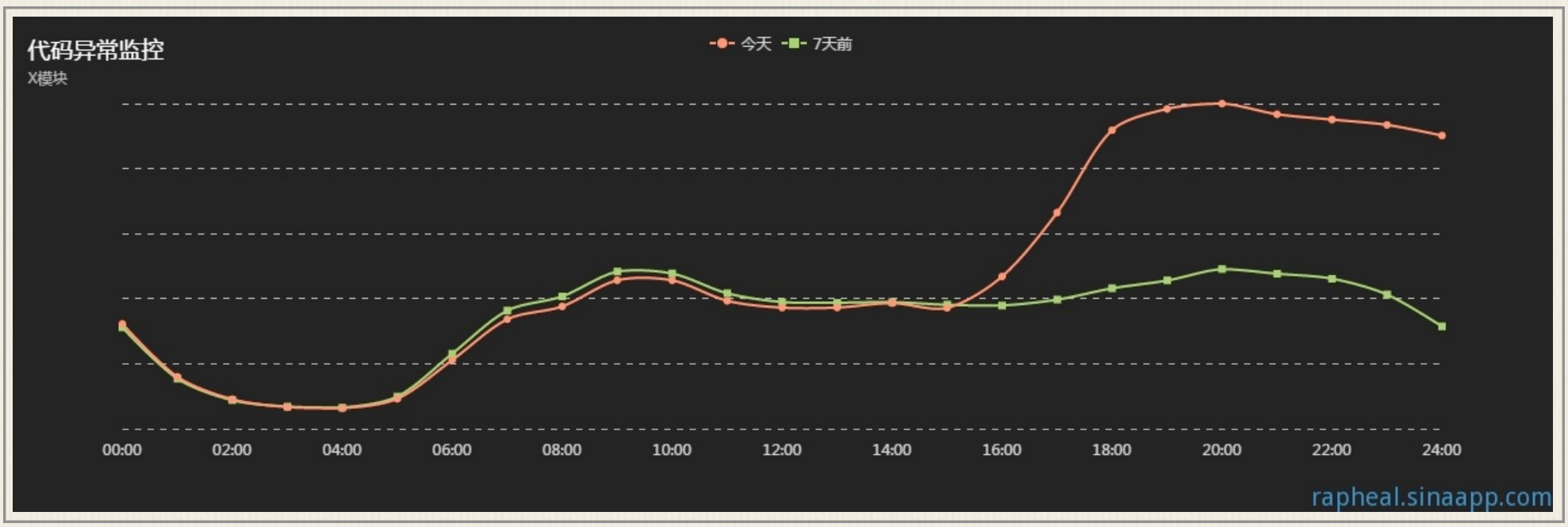
假如我们能监控到图文页一天异常发生量，我们在发送异常的时候往服务器上报异常信息，服务器就统计出每一分钟异常的总数：



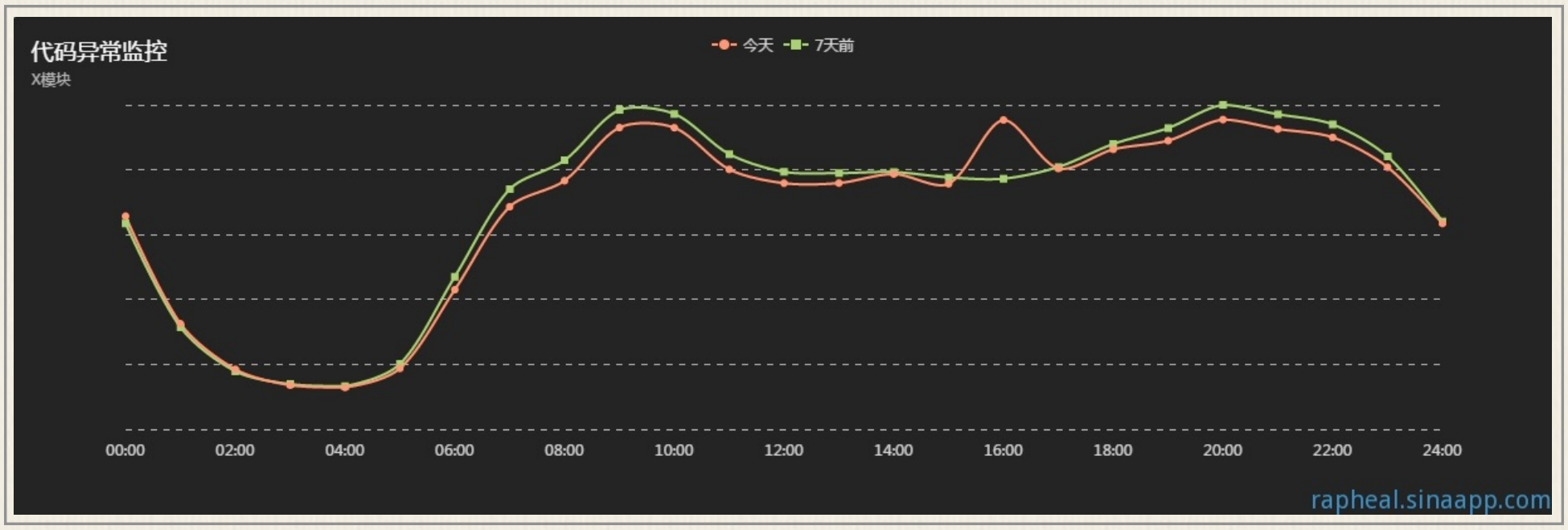
那我们可以非常直观地看到发生异常的量，例如上图的例子就可以看出，今天跟上周的异常量是吻合的，没有特别的突增或者突减（当然异常减少是我们需要 去做到的！但是后边会讲到如用户浏览器的Javascript插件如果运行时出错，也会被我们捕捉到，所以实际上很难把异常数清零）。

上线

如果有了上边的监控，我们能做什么呢？回到15:00，我上线后，后台收到了非常多的异常上报，于是：



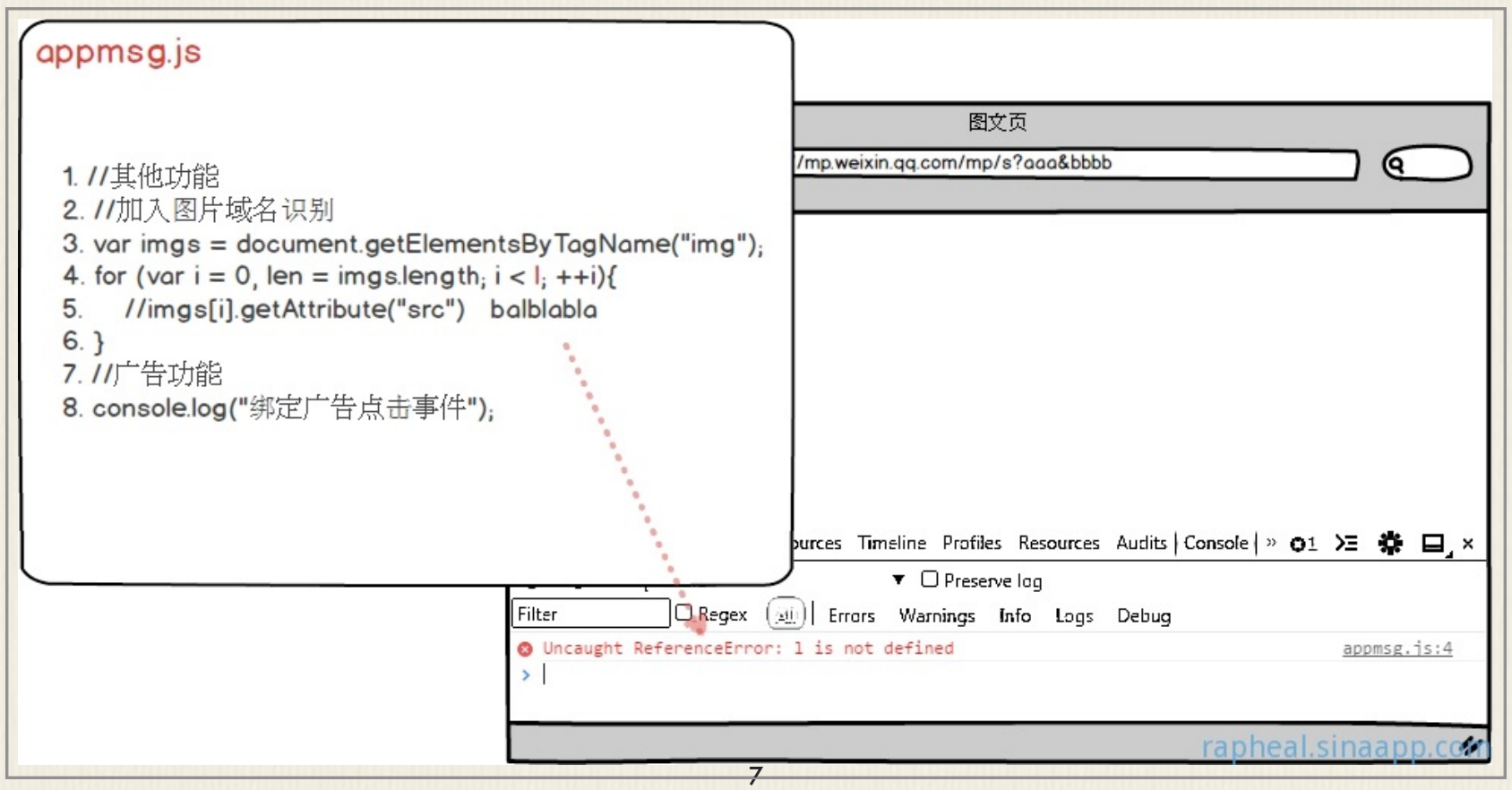
这时候后台发现监控曲线已经偏离7天前（或者昨天）超过一定的百分比（或者数量），立马发短信发微信给负责人，在用户投诉之前，负责人已经知道这个异常现象的发生，只要及时做修复，就可以恢复正常：



在上线的时候会有一个异常数的飙升，在修复后，异常数恢复到正常范围内。

如何检测前端异常

那剩下的问题就是如何检测前端的异常，先看看刚刚那段代码在浏览器的出错展现：



一般语法错误以及运行时错误，浏览器都会在console里边体现出错误信息，以及出错的文件，行号，堆栈信息。

来到这里，我们要定义一下本文说到的前端代码异常是什么意思。前端代码异常指的是以下两种情况：

1. JS脚本里边存着语法错误；
2. JS脚本在运行时发生错误。

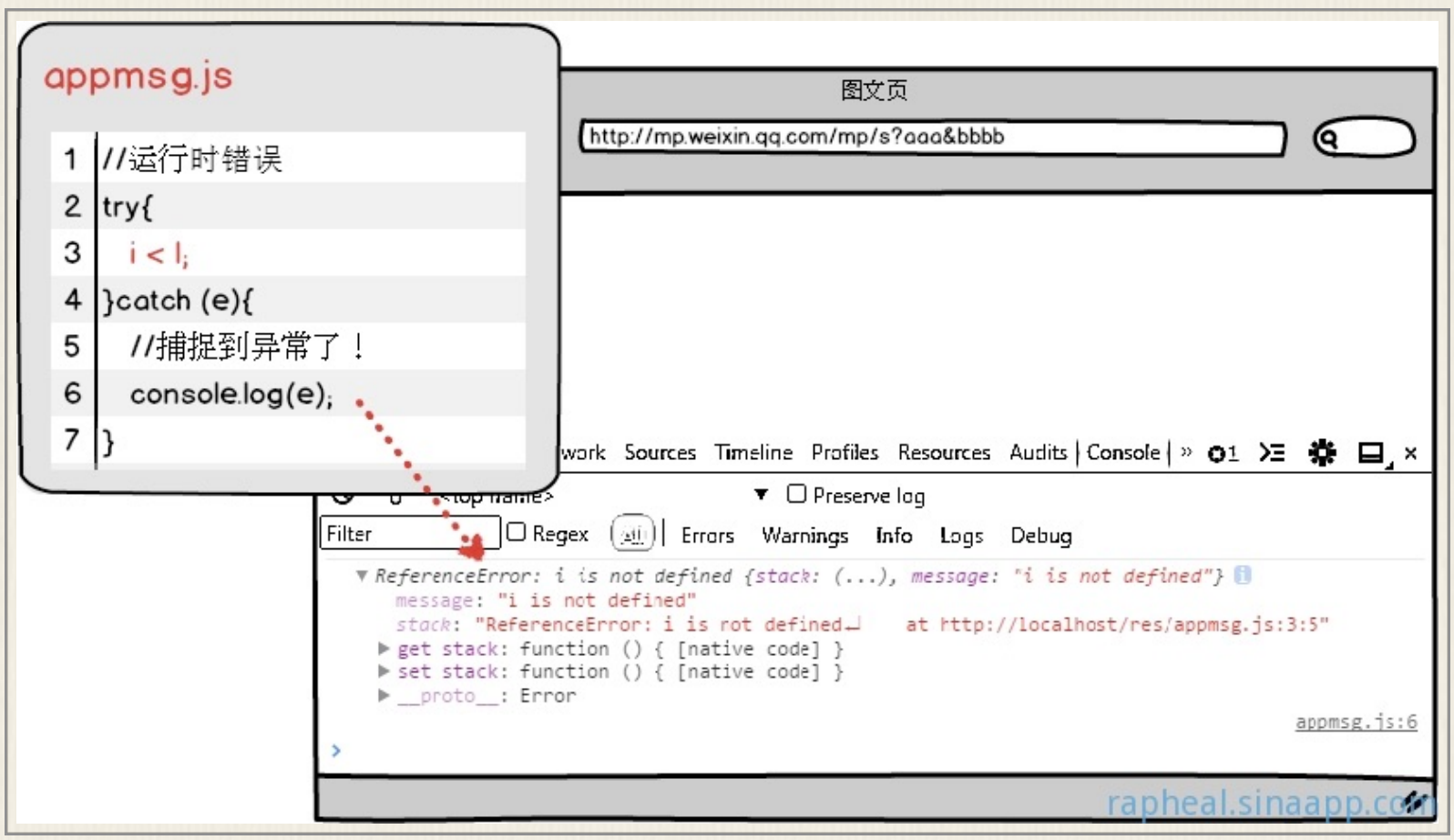
有什么方法可以抓到这个错误，有两个方案：

1. try, catch方案。你可以针对某个代码块使用try,catch包装，这个代码块运行时出错时能在catch块里边捕捉到。
2. window.onerror方案。也可以通过window.addEventListener(“error”, function(evt){})，这个方法能捕捉到语法错误跟运行时错误，同时还能知道出错的信息，以及出错的文件，行号，列号。

上边只是简单的说了一下方案，我在接下来的2个小节来讨论一下两个方案实现细节。

try,catch

我们可以通过对代码块加入一个try,catch块来抓出错信息：



从console可以看到，try,catch能够知道出错的信息，并且也有堆栈信息可以知道在哪个文件第几行第几列发生错误。

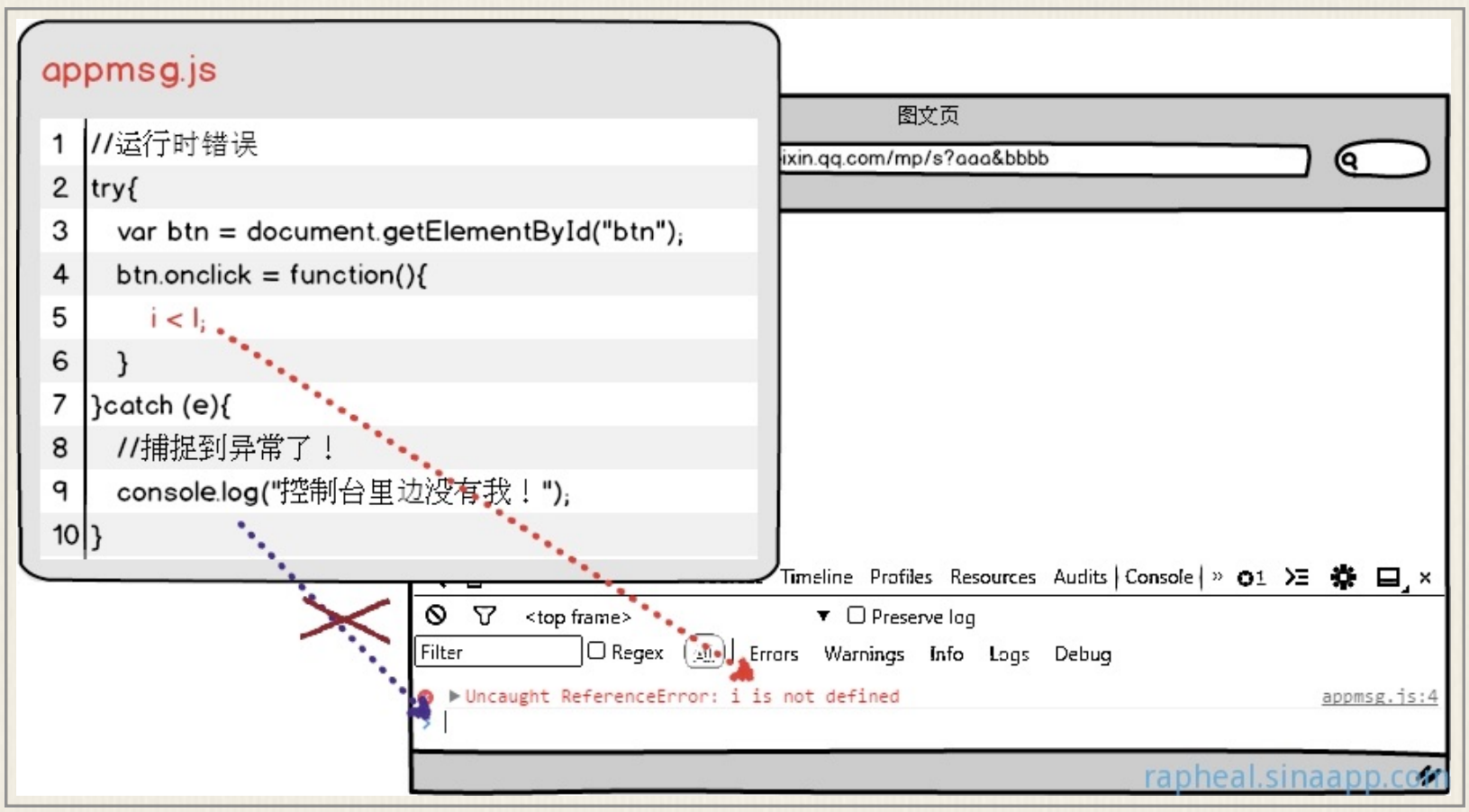
但是try,catch的方案有2个缺点：

1. 没法捕捉try,catch块，当前代码块有语法错误，JS解释器压根都不会执行当前这个代码块，所以也就没办法被catch住；

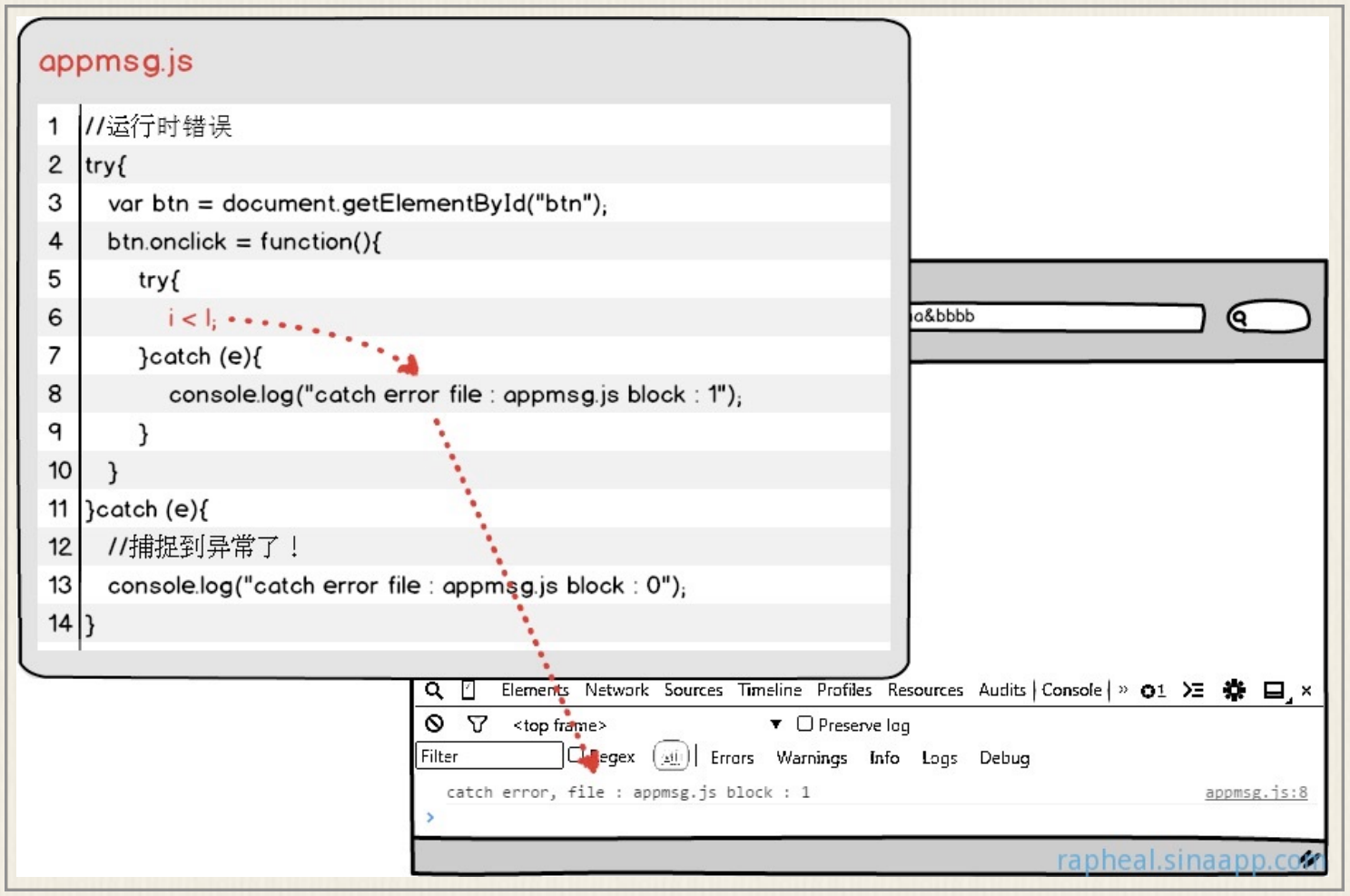
2. 没法捕捉到全局的错误事件，也即是只有try,catch的块里边运行出错才会被你捕捉到，这里的块你要理解成一个函数块。

关于第一个缺点，我们没有任何解决办法，原因上边说了，但是一般语法阶段我们是能在开发阶段/或者用工具检测到的，于是乎它就被忽略了。

第二个缺点应该怎么理解呢？try, catch只能捕捉到当前执行流里边的运行错误，对于异步回调来说，是不属于这个try,catch块的：

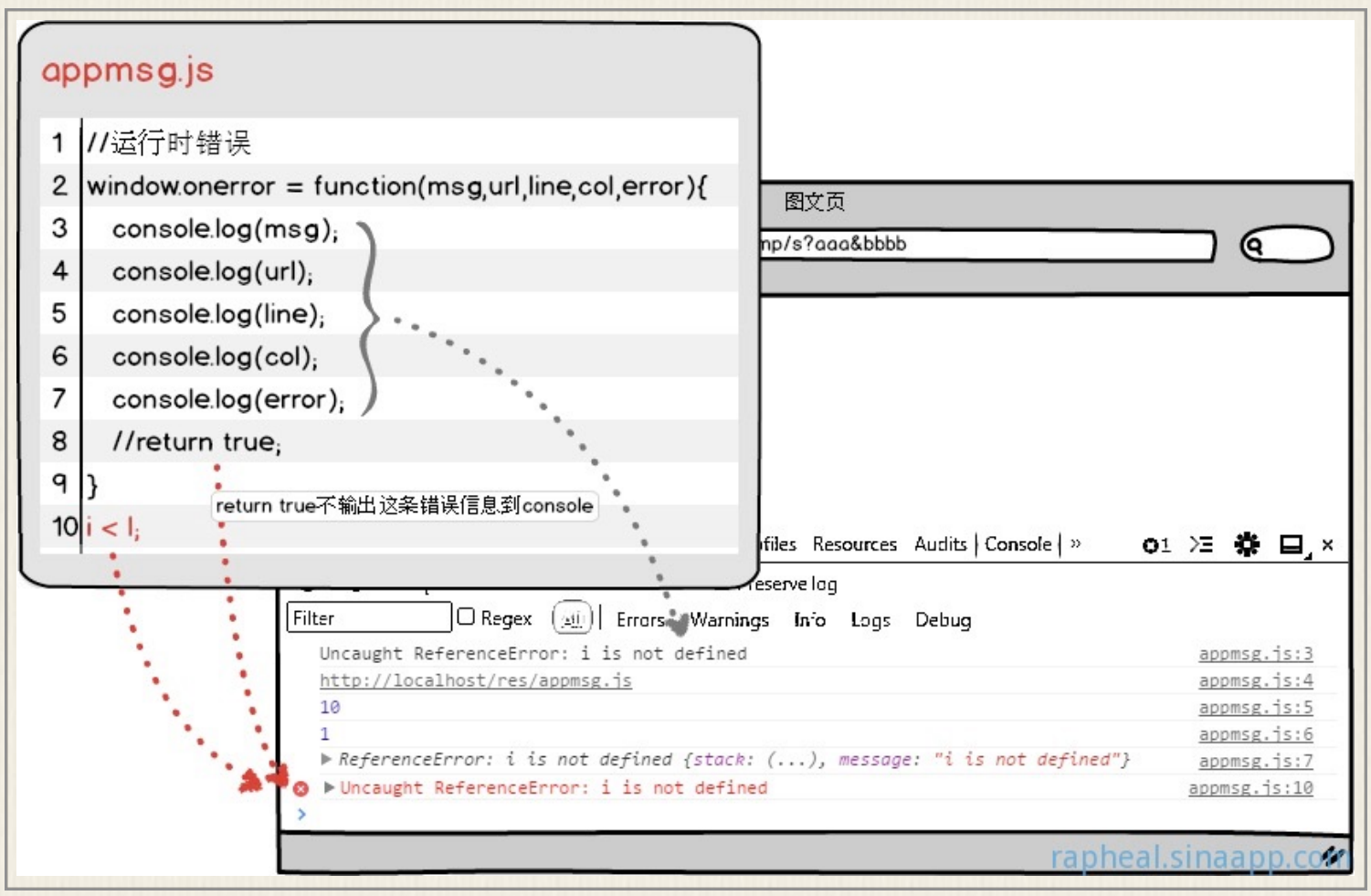


我们可以怎么去改进这个方案，我利用了uglifyjs的词法语法分析，再uglifyjs 最后输出压缩文件的时候往文件块以及function块加入了try,catch，同时为每个块加入编号，这样出错的时候我们在日志里边就能看到是哪个块有问题。

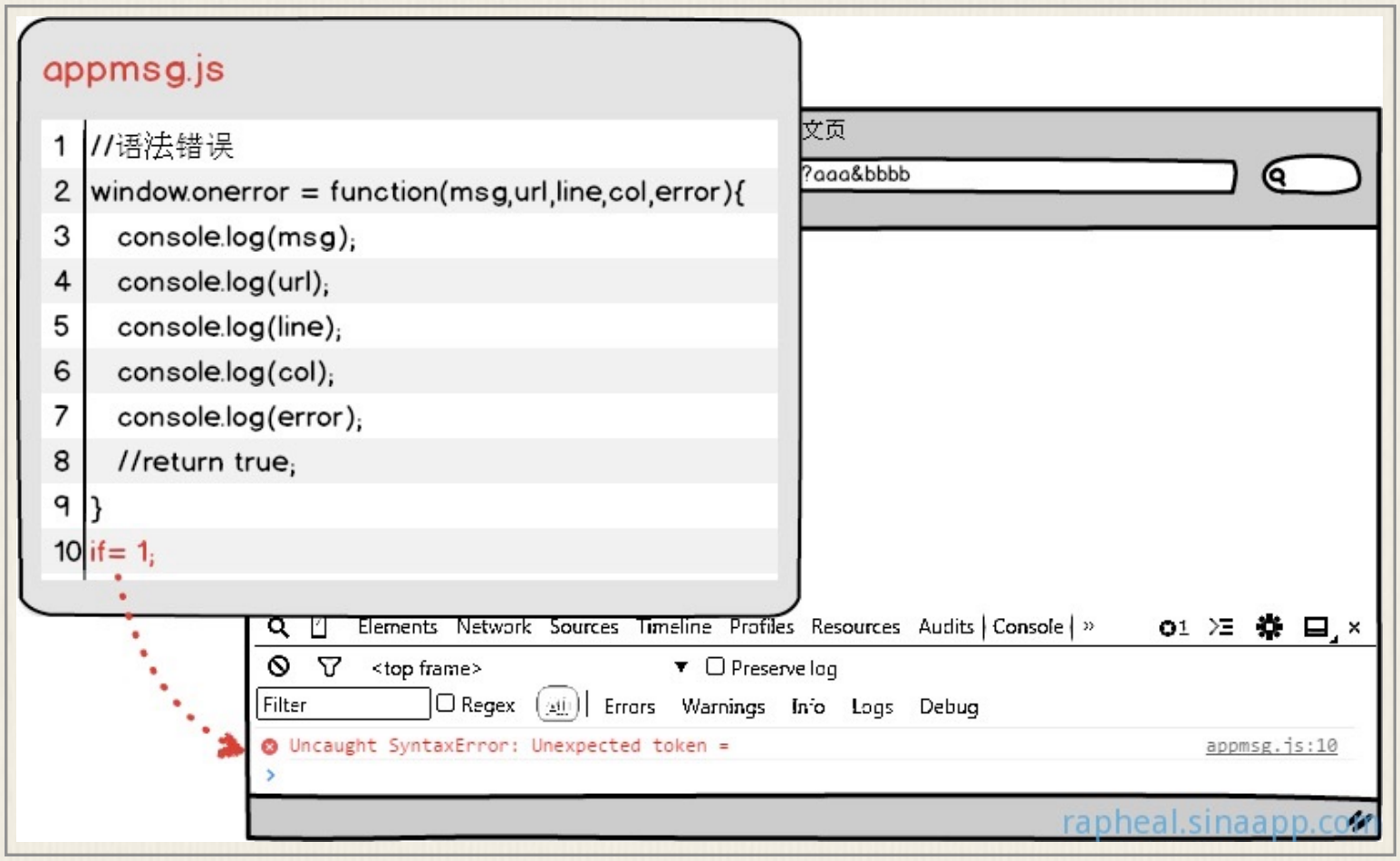


window.onerror

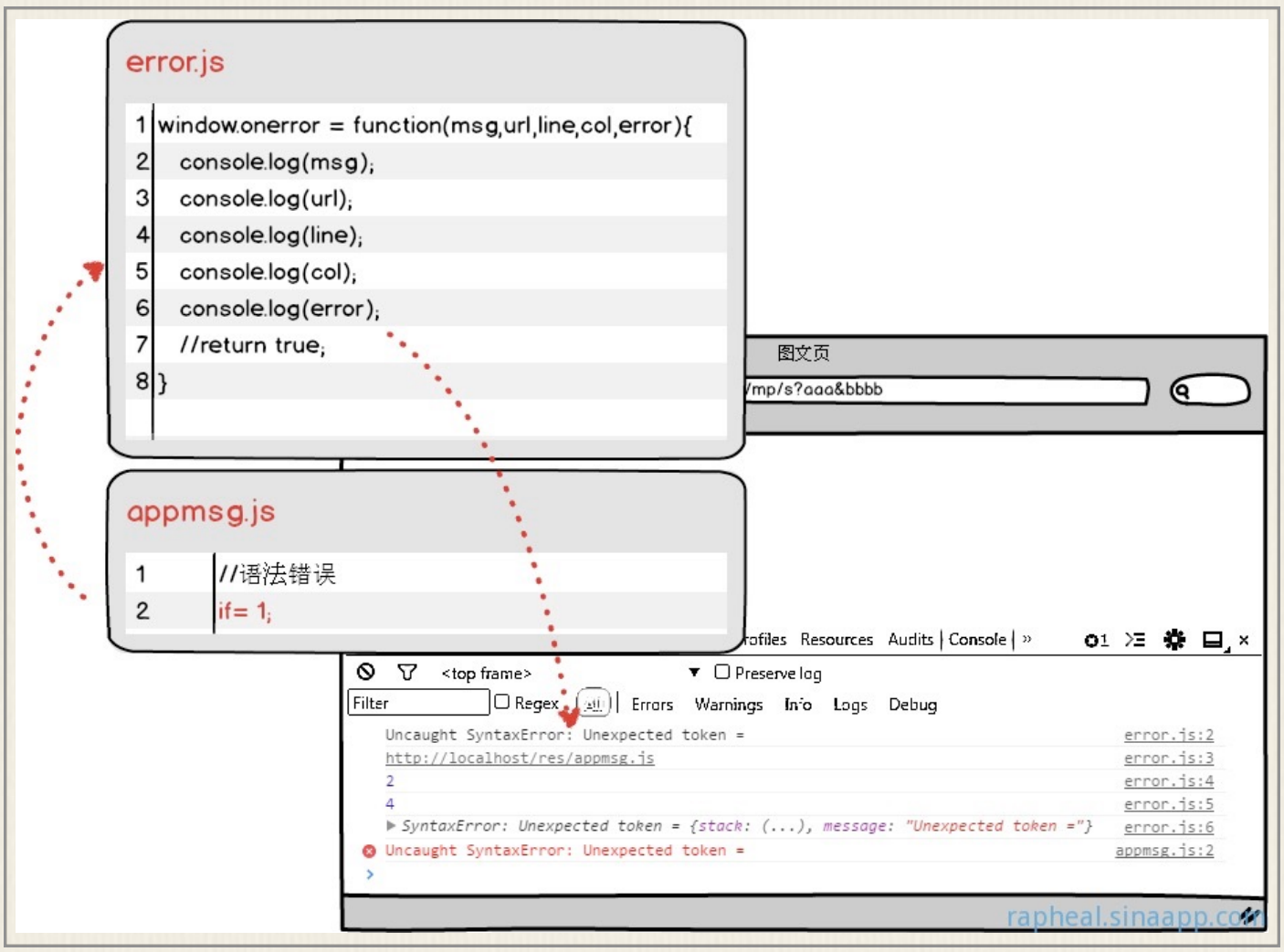
window.onerror一样可以拿到出错的信息以及文件名、行号、列号，还可以在window.onerror最后return true让浏览器不输出错误信息到控制台。



window.onerror能捕捉到语法错误，但是语法出错的代码块不能跟window.onerror在同一个块（语法都没过，更别提window.onerror会被执行了）



只要把window.onerror这个代码块分离出去，并且比其他脚本先执行（注意这个前提！）即可捕捉到语法错误。



对于跨域的JS资源，window.onerror拿不到详细的信息，需要往资源的请求添加额外的头部。

静态资源请求需要加多一个Access-Control-Allow-Origin头部，同时script引入外链的标签需要加多一个crossorigin的属性。经过这样折腾后一样能获取到准确的出错信息。

最终方案

我们先来比较两个方案各自的特点。

try,catch的方案有如下特点：

1. 无法捕捉到语法错误，只能捕捉运行时错误；
2. 可以拿到出错的信息，堆栈，出错的文件、行号、列号；
3. 需要借助工具把所有的function块以及文件块加入try,catch，可以在这个阶段打入更多的静态信息。

window.onerror的方案有如下特点：

1. 可以捕捉语法错误，也可以捕捉运行时错误；

2. 可以拿到出错的信息，堆栈，出错的文件、行号、列号；

3. 只要在当前页面执行的js脚本出错都会捕捉到，例如：浏览器插件的javascript、或者flash抛出的异常等。

4. 跨域的资源需要特殊头部支持。

window.onerror的方法要比try,catch方法更加完善，我们允许少量由于插件带来的脚本错误，最后window.onerror实现的方式是：

```
window.onerror = function(msg,url,line,col,error){
    //没有URL不上报！上报也不知道错误
    if (msg != "Script error." && !url){
        return true;
    }
    //采用异步的方式
    //我遇到过在window.onunload进行ajax的堵塞上报
    //由于客户端强制关闭webview导致这次堵塞上报有Network Error
    //我猜测这里window.onerror的执行流在关闭前是必然执行的
    //而离开文章之后的上报对于业务来说是可丢失的
    //所以我把这里的执行流放到异步事件去执行
    //脚本的异常数降低了10倍
    setTimeout(function(){
        var data = {};
        //不一定所有浏览器都支持col参数
```

```
col = col || (window.event && window.event.errorCharacter) || 0;

data.url = url;
data.line = line;
data.col = col;
if (!!error && !!error.stack){
    //如果浏览器有堆栈信息
    //直接使用
    data.msg = error.stack.toString();
}else if (!!arguments.callee){
    //尝试通过callee拿堆栈信息
    var ext = [];
    var f = arguments.callee.caller, c = 3;
    //这里只拿三层堆栈信息
    while (f && (--c>0)) {
      ext.push(f.toString());
      if (f  === f.caller) {
          break;//如果有环
      }
      f = f.caller;
    }
    ext = ext.join(",");
    data.msg = ext;
}
```

```
        //把data上报到后台!
    },0);

    return true;
};
```

后话

通过部署代码异常监控之后，不仅仅用以监控平时上线的异常，同时还发现了不少旧有代码的错误。
例如以下代码:

```
src = img.getAttribute("src");
src.indexOf("http://rapheal.sinaapp.com/");
```

在某些情况下，文章里边的src可能是null，导致这里调用null的indexOf方法发生异常。
也检测到微信webview里边的一些客户端抛出的异常，可以进一步让客户端开发的同事去做bug fix。

上线的稳定性不仅仅依托于代码异常的监控，代码异常监控只能监控到你代码的健康性，而很多时候业务的稳定还需要监控一些业务数据，例如昨天有 1000个人点击了关注按钮，今天上线后突然变成了300人点击，除非你很清楚你上线的行为是会导致点击数下降，否则我们就应该重新审查这次上线是否存在 问题，必要时还应该回退这次上线。

原文链接：http://rapheal.sinaapp.com/2014/11/06/javascript-error-monitor/

关于C++14：你需要知道的新特性

译者：Rachel

使C++更加安全和更加方便的有用新特性

今年8月，经过投票， C++14标准获得一致通过。目前唯一剩下的工作是ISO进行C++标准的正式发布。在本文中，我关注的是新标准中的几个重要点，展示了即将到来的改变会如何影响你的编程方式，特别是在使用被现代C++称之为习语和范型的特性时。

C++标准委员会决心使标准制定过程比过去10年更加快速。这意味着，距上一个标准（即C++11）仅3年的C++14是一次相对较小的发布。这远 非一个令人失望的消息，恰恰相反，这对程序员来说是个好消息。因为这样的话，开发人员能够实时地跟上新特性。所以，今天你就可以开始使用C++14的新特 性了—而且，如果你的工具链足够灵活的话，你几乎可以使用全部新特性了。

目前你可以从这里得到标准草案的一份免费副本。遗憾的是，当最终版本的标准发布时，ISO会进行收费。

缩短标准发布的时间间隔可以帮助编译器作者更实时地跟上语言变化。仅隔三年就再次发布，需要调整以适应的变化也就更少。

本文的例子主要在clang 3.4上测试过，clang 3.4覆盖了大多数C++14的新特性。目前，g++对新特性的覆盖更少一些，而Visual C++似乎落后更多。

C++14：重大变化

接下来，本文将说明对程序员编码工作会有重大影响的C++14特性，在给出实例的同时，还讨论了何时何地因何使用这些特性。

返回类型推导

在这次发布中，关键字auto的作用扩大了。C++语言本身仍然是类型安全的，但是类型安全的机制逐渐改由编译器而不是程序员来实现。

在C++11中，程序员最初使用auto是用于声明。这对于像迭代器的创建之类尤其有用，因为完整的正确的类型名可能长得可怕。使用了auto的C++代码则易读得多：

for (auto ii = collection.begin() ; ...

在C++14中，auto的使用在好几个方面得到了扩展。其中之一便是意义非凡的返回类型推导。在一个函数中编写如下一行代码：这段代码依然完全地 是类型安全的，因为编译器知道begin()在上下文中应该返回什么类型。因此，ii的类型是毫无疑问的，并且在使用ii的每个地方，编译器都会进行检查。

return 1.4;

对于程序员和编译器来说，很显然，函数返回的是double类型。因此在C++14中，程序员可以用auto代替double来定义函数返回类型：

auto getvalue() {

这个新特性需要注意的一个细节也是相当容易理解的。那就是，如果一个函数有多个返回路径，那么每个返回路径返回的值需要具有相同的类型。

```
auto f(int i)
{
  if ( i < 0 )
    return -1;
  else
    return 2.0
}
```

上面这段代码似乎显然应该推导出返回类型是double，但是C++14禁止这种带歧义性的使用。对于上述代码，编译器会报错：

```
error_01.cpp:6:5: error: 'auto' in return type deduced as 'double' here but deduced as 'int' in
      earlier return statement
    return 2.0
    ^
1 error generated.
```

为C++程序增加推导返回类型这一特性有诸多很好的理由。第一个理由是，有时候需要返回相当复杂的类型，例如，在STL容器中进行搜索时可能需要返 回迭代器类型。auto使函数更易编写，更具可读性。第二个（可能不那么明显的）理由是，auto的使用能够增强你的重构能力。考虑以下程序：

```
#include <iostream>
#include <vector>
#include <string>

struct record {
   std::string name;
   int id;
};

auto find_id(const std::vector<record> &people,
```

```
                const std::string &name)
{
  auto match_name = [&name](const record& r) -> bool {
    return r.name == name;
  };
  auto ii = find_if(people.begin(), people.end(), match_name );
  if (ii == people.end())
    return -1;
  else
    return ii->id;
}

int main()
{
  std::vector<record> roster = { {"mark",1},
                                 {"bill",2},
                                 {"ted",3}};
  std::cout << find_id(roster,"bill") << "\n";
  std::cout << find_id(roster,"ron") << "\n";
}
```

在这个例子中，使用auto代替int作为find_id()函数的返回类型并不能节省多少脑细胞J。但是，考虑一下，如果我决定重构record结构，将会发生什么。或许我想用一个新的类型GUID而不是一个整型来标识record对象中的人：

```
struct record {
   std::string name;
   GUID id;
};
```

record对象的变化将引起包括函数返回类型在内的一系列级联变化。但是，如果我在函数中使用了自动的返回类型推导，那么编译器将默默地为我进行这些修改。

任何有过大型项目工作经验的C++程序员都应该很熟悉这个问题–对单一数据结构的修改可能引起代码库看似无穷无尽的迭代：修改变量，修改参数，修改返回类型。auto的增加使用对减少这种工作贡献不小。

注意在上述例子及本文的余下部分，我创建并使用有名的lambda。我猜想，大多数用户在std::find_if()这样的函数中都是把 lambda定义为匿名的内联对象的，这确实是非常方便的方式。由于浏览器的页面宽度有限，我认为把lambda的定义和使用分开能够使读者通过浏览器阅 读代码比较容易。因此，这并不是各位读者一定应该仿效的方式，读者们只是应该感激这样使代码更加易读–特别是，当你是一位缺乏lambda使用经验的读者 时。

说回auto，使用auto作为返回类型带来的一个直接推论是其分身decltype(auto)的实现，以及它在类型推导时将遵循的规则。像下面的代码片段展示的一样，现在你可以使用它自动地捕获类型信息：

```
template<typename Container>
struct finder {
  static decltype(Container::find) finder1 = Container::find;
  static decltype(auto) finder2 = Container::find;
};
```

泛型Lambdas

auto悄悄潜伏的另一个地方是lambda参数的定义。使用auto类型声明来定义lambda参数等价于放松限制地创建模板函数。基于推导出的参数类型，lambda将进行特定的实例化。

这方便了可重用于不同上下文的lambda的创建。在下文的简单例子中，我创建了一个lambda，用来作为一个标准库函数的谓词函数。在C++11中，我需要明确地实例化一个lambda用于整数的相加，再实例化另一个lambda用于字符串的相加。

有了泛型lambda后，我可以只定义一个带有泛型参数的lambda。尽管泛型lambda在语法上没有包含关键字template，但是很显然，它仍是C++泛型编程的进一步延展。

```
#include <iostream>
#include <vector>
#include <string>
#include <numeric>

int main()
{
  std::vector<int> ivec = { 1, 2, 3, 4};
  std::vector<std::string> svec = { "red",
                                    "green",
                                    "blue" };
  auto adder  = [](auto op1, auto op2){ return op1 + op2; };
  std::cout << "int result : "
            << std::accumulate(ivec.begin(),
```

```
                        ivec.end(),
                        0,
                        adder )
        << "\n";
  std::cout << "string result : "
        << std::accumulate(svec.begin(),
                        svec.end(),
                        std::string(""),
                        adder )
        << "\n";
  return 0;
}
```

上述代码产生以下输出：

```
int result : 10
string result : redgreenblue
```

即使你实例化匿名的内联lambda，采用泛型参数仍然是有用的，原因我已在前文中讨论过，这里再复述一下–当你的数据结构改变时，或者API中获 取签名的函数修改时，泛型lambda将在重新编译时自行调整而不需要重写代码。使用泛型参数的匿名内联lambda例子如下所示：

```
std::cout << "string result : "
        << std::accumulate(svec.begin(),
                        svec.end(),
                        std::string(""),
                        [](auto op1,auto op2){ return op1+op2; } )
```

```
        << "\n";
```

可初始化的Lambda捕获

在C++11中，我们不得不开始适应lambda capture这一概念。其声明指导编译器进行closure的创建：closure是一个由lambda定义的函数的实例，同时，它绑定了定义于lambda作用域之外的变量。

在上文有关推导返回类型的示例中，定义了一个lambda，它捕获变量name，该变量被作为一个搜索字符串的谓词函数的源：

```
auto match_name = [&name](const record& r) -> bool {
    return r.name == name;
 };
 auto ii = find_if(people.begin(), people.end(), match_name );
```

这种特殊的捕获使lambda能够访问到引用变量。捕获也可以通过值来完成。在这两种情形中，变量的使用符合C++一贯的方式–通过值捕获时lambda操作的是变量的本地副本，而通过引用捕获则意味着lambda作用于来自其作用域之外的变量实例本身。

这一切都OK，但同时也带来了一些限制。我认为，C++标准委员会觉得需要特别强调的一点是，不能使用move-only语法来初始化捕获的变量。

这说明什么呢？如果我们想把lambda作为一个参数的sink（接收器），我们会使用move语法来捕获其作用域之外的变量。作为一个例子，考虑 一下如何得到一个lambda，它接收具有move-only特点的unique_ptr对象。首先，尝试通过值捕获将以失败告终：

```
std::unique_ptr<int> p(new int);
  *p = 11;
```

```
auto y = [p]() { std::cout << "inside: " << *p << "\n";};
```

这段代码产生编译错误是因为unique_ptr不会生成拷贝构造函数–unique_ptr本身就是为禁止拷贝而生的。

修改代码通过引用捕获p能够编译通过，但是这并不能达到期望的效果，我们的初衷是通过移动变量的值到本地拷贝来接收变量值。最终，创建一个局部变量并在通过引用捕获时调用std::move()能够达到目的，但是其效率略低。

修改捕获子句的语法可以解决效率低的问题。现在，不仅仅可以声明一个捕获变量，还可以进行初始化。作为标准中的一个例子，简单情形下的使用看起来像这样：

```
auto y = [&r = x, x = x+1]()->int {...}
```

它捕获x的副本同时实现对x的增量操作。这个例子很容易理解，但是我不确定它是否能够捕获这种新语法下的move-only变量的值。一个利用了这个新语法的用例如下所示：

```
#include <memory>
#include <iostream>

int main()
{
  std::unique_ptr<int> p(new int);
  int x = 5;
  *p = 11;
  auto y = [p=std::move(p)]() { std::cout << "inside: " << *p << "\n";};
  y();
```

```
  std::cout << "outside: " << *p << "\n";
  return 0;
}
```

在这个例子中，捕获的变量值p通过move语法进行初始化，在不需要声明一个局部变量的情况下有效地接收了指针。

```
inside: 11
Segmentation fault (core dumped)
```

这个恼人的结果正是你所期望的–代码在变量p已经被捕获并移动到lambda后试图解引用它（这当然会导致错误）。

[[deprecated]]属性

当我第一次在Java中见到deprecated属性的使用时，我承认我有点嫉妒这门语言。对大多数程序员来说，代码陈旧是个大问题。（有因删除代码而被称赞过吗？反正我从来没有。）这个新属性提供了解决这个问题的系统方法。

它的用法方便又简单—只需要把[[deprecated]]标签放到声明的前面即可—可以是类，变量，函数，或者其他一些实体的声明。结果看起来像这样：

```
class
[[deprecated]] flaky {
};
```

当程序中使用了过时的实体时，编译器的反应是把它留给开发人员。显然，大多数人会希望在需要时看到某种形式的警告，同时在不需要时也能够关掉警告。clang3.4中有一个例子，当实例化一个过时的类时给出了警告：

dep.cpp:14:3: warning: 'flaky' is deprecated [-Wdeprecated-declarations]

flaky f;

^

dep.cpp:3:1: note: 'flaky' declared here

flaky {

^

你可能已经注意到，C++的attribute-tokens语法看起来似乎有点不常见。包含[[deprecated]]的属性列表，被放在class，enum等关键字之后，实体名之前。

这个标签具有包括消息参数的另一种形式。同样地，如何处理该消息取决于开发人员。显然，clang3.4直接忽略了该消息。因为，如下代码片段的输出中并不包含错误消息：

```
class
[[deprecated]] flaky {
};

[[deprecated("Consider using something other than cranky")]]
int cranky()
{
   return 0;
}
```

```
int main()
{
  flaky f;
  return cranky();
}
```

```
dep.cpp:14:10: warning: 'cranky' is deprecated [-Wdeprecated-
declarations]
  return cranky();
         ^
dep.cpp:6:5: note: 'cranky' declared here
int cranky()
    ^
```

二进制常量和单引号用作数字分位符

这两个新特性并不惊天动地，但它们确实代表了好的语法改进。语言中像这样的持续小改进可以提高代码的可读性并因此而减少bug数量。

除了原有的十进制、十六进制和比较不常用的八进制表示方法之外，C++程序员现在还可以使用二进制表示常量了。二进制常量以前缀0b（或0B）开头，二进制数字紧随其后。

在英美两国，在写数字时，我们习惯于使用逗号作为数字的分隔符，如：$1,000,000。这些数字分隔符纯为方便读者，它提供的语法线索使我们的大脑在处理长串的数字时更加容易。

基于完全相同的原因，C++标准委员会为C++语言增加了数字分隔符。数字分隔符不会影响数字的值，它们的存在仅仅是为了通过分组使数字的读写更容易。

使用哪个字符来表示数字分隔符呢？在C++中，几乎每个标点字符都已经有特定的用途了，因此并没有明显的选择。最终的结果是使用单引号字符，这使得 百万美元在C++中写作1’000’000.00。记住，分隔符不会对常量的值有任何影响，因此，1’0’00’0’00.00也是表示百万。

下面是一个结合了这两种新特性的例子：

```
#include <iostream>

int main()
{
  int val = 0b11110000;
  std::cout << "Output mask: "
            << 0b1000'0001'1000'0000
            << "\n";
  std::cout << "Proposed salary: $"
            << 300'000.00
            << "\n";
  return 0;
}
```

这段代码的输出毫不令人吃惊：

```
Output mask: 33152
Proposed salary: $300000
```

其他

C++14规范中的其他特性并不需要如此多的阐释。

变量模板就是将模板扩展到变量。用滥了的例子是变量模板pi<T>的实现。当T表示double类型时，变量返回3.14。表示int 类型时，返回3。表示std::string类型时，则可能返回字符串”3.14”或者”pi”。当<limits>头文件写好的时候，这将 是一个很好的特性。

变量模板的语法及语义与类模板几乎是相同的，所以，即使不进行任何额外的学习，使用它们也应该是没有问题的（如果你已经了解了类模板的话）。

constexpr函数的限制被放松了。现在允许在case语句，if语句，循环语句等语句中进行多处返回了。这扩展了可在编译期间完成的事情的范围，增加可在编译期间完成的事情这一趋势在模板被引入后发展得尤其迅速。

其他的小特性包括可指定大小的资源回收函数和一些语法整理。

接下来

C++标准委员会明显感受到了压力，正在通过改进来保持C++语言与时俱进。在这个十年期中，他们已经在至少一个（即C++17）以上的标准上进行努力了。

也许更有趣的是，几个衍生组织的创立，这些组织可以创建技术规范文档。这些文档不会提升为标准，但是它们会发表并获得ISO标准委员会的认可。根据推测，这些事务将以更快的速度得到推进。这些组织当前工作的八大领域包括以下方面：

- 文件系统
- 并发性
- 并行性
- 网络
- C++的AI概念（Artificial Intelligence，人工智能）–一直处于规范中。

这些技术规范的成功与否取决于其是否被采纳和使用。如果我们发现所有开发人员都跟随它们，那么这种进行标准化的新途径就算成功了。

这些年来C/C++发展良好。现代C++（或许以C++11作为开始）在保持性能的同时，在使C++语言更加易用更加安全方面取得了引人注目的进 展。对于某些类型的工作，你很难找出C/C++之外的任何合理替代品。C++14并未做出C++11版本中那样的大改进，但是它把语言保持在一条很好的路上。如果C++标准委员会在未来十年保持其目前的效率，那么C++应该能够继续作为当性能被定为目标时的首选语言。

译文链接：http://blog.jobbole.com/79228/

原文链接：http://www.drdobbs.com/cpp/the-c14-standard/240169034

Leveldb 实现原理

作者：朗格科技

LevelDb日知录之一：初识LevelDb

说起LevelDb也许您不清楚，但是如果作为IT工程师，不知道下面两位大神级别的工程师，那您的领导估计会Hold不住了：Jeff Dean和Sanjay Ghemawat。这两位是Google公司重量级的工程师，为数甚少的Google Fellow之二。

Jeff Dean其人：

http://research.google.com/people/jeff/index.html

Google大规模分布式平台Bigtable和MapReduce主要设计和实现者。

Sanjay Ghemawat其人：

http://research.google.com/people/sanjay/index.html

Google大规模分布式平台GFS，Bigtable和MapReduce主要设计和实现工程师。

LevelDb就是这两位大神级别的工程师发起的开源项目，简而言之，LevelDb是能够处理十亿级别规模Key-Value型数据持久性存储的C++ 程序库。正像上面介绍的，这二位是Bigtable的设计和实现者，如果了解Bigtable的话，应该知道在这个影响深远的分布式存储系统中有两个核心 的部分：Master Server和Tablet Server。其中Master Server做一些管理数据的存储以及分布式调度工作，实际的分布式数据存储以及读写操作是由Tablet Server完成的，而LevelDb则可以理解为一个简化版的Tablet Server。

LevelDb有如下一些特点：

首先，LevelDb是一个持久化存储的KV系统，和Redis这种内存型的KV系统不同，LevelDb不会像Redis一样狂吃内存，而是将大部分数据存储到磁盘上。

其次，LevleDb在存储数据时，是根据记录的key值有序存储的，就是说相邻的key值在存储文件中是依次顺序存储的，而应用可以自定义key大小比较函数，LevleDb会按照用户定义的比较函数依序存储这些记录。

再次，像大多数KV系统一样，LevelDb的操作接口很简单，基本操作包括写记录，读记录以及删除记录。也支持针对多条操作的原子批量操作。

另外，LevelDb支持数据快照（snapshot）功能，使得读取操作不受写操作影响，可以在读操作过程中始终看到一致的数据。

除此外，LevelDb还支持数据压缩等操作，这对于减小存储空间以及增快IO效率都有直接的帮助。

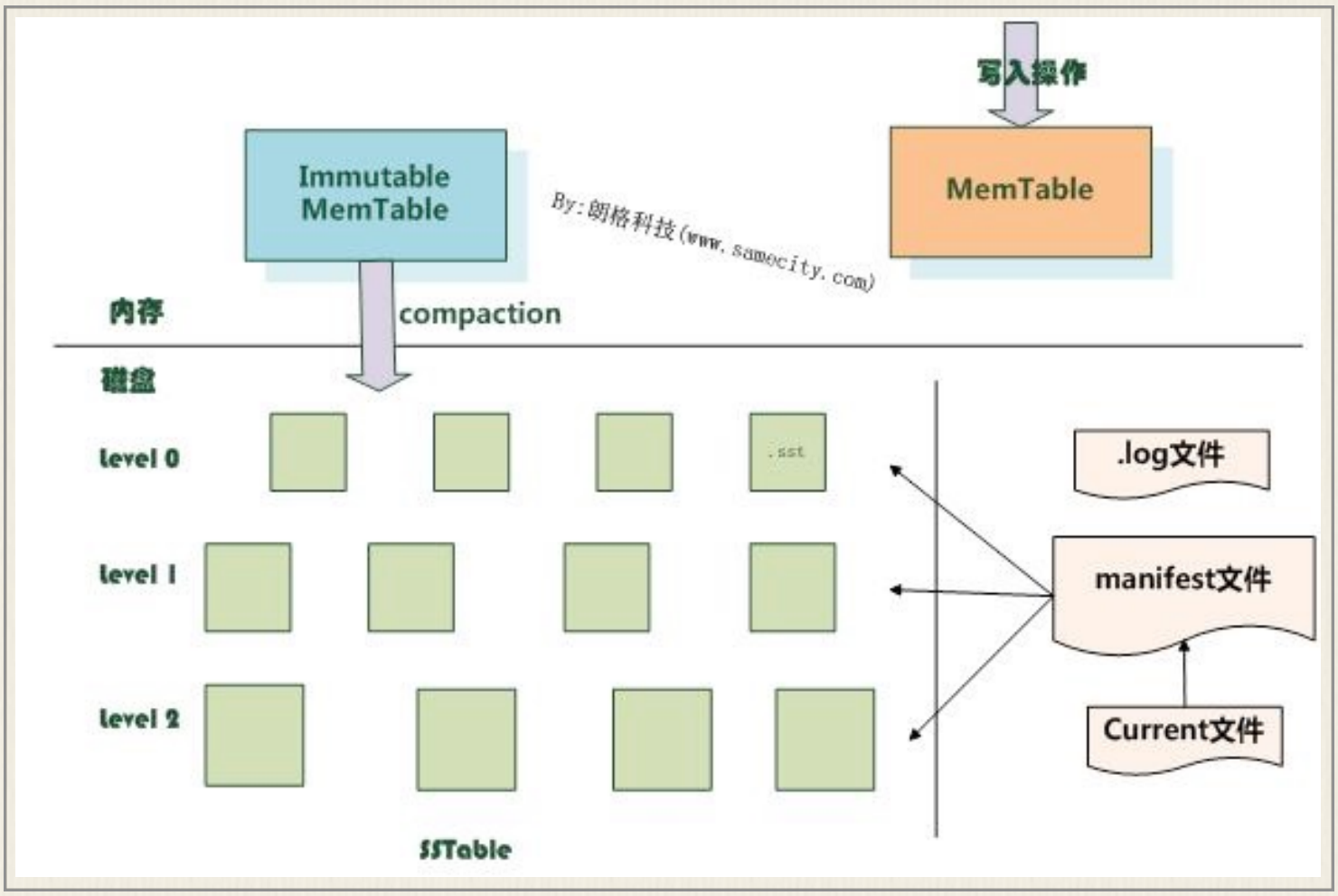
LevelDb性能非常突出，官方网站报道其随机写性能达到40万条记录每秒，而随机读性能达到6万条记录每秒。总体来说，LevelDb的写操作要大 大 快于读操作，而顺序读写操作则大大快于随机读写操作。至于为何是这样，看了朗格科技后续推出的LevelDb日知录，估计您会了解其内在原因。

LevelDb日知录之二整体架构

LevelDb本质上是一套存储系统以及在这套存储系统上提供的一些操作接口。为了便于理解整个系统及其处理流程，我们可以从两个不同的角度来看待 LevleDb：静态角度和动态角度。从静态角度，可以假想整个系统正在运行过程中（不断插入删除读取数据），此时我们给LevelDb照相，从照片可以 看到之前系统的数据在内存和磁盘中是如何分布的，处于什么状态等；从动态的角度，主要是了解系统是如何写入一条记录，读出一条记录，删除一条记录的，同时 也包括除了这些接口操作外的内部操作比如compaction，系统运行时崩溃后如何恢复系统等等方面。

本节所讲的整体架构主要从静态角度来描述，之后接下来的几节内容会详述静态结构涉及到的文件或者内存数据结构，LevelDb日知录后半部分主要介绍动态视角下的LevelDb，就是说整个系统是怎么运转起来的。

LevelDb作为存储系统，数据记录的存储介质包括内存以及磁盘文件，如果像上面说的，当LevelDb运行了一段时间，此时我们给LevelDb进行透视拍照，那么您会看到如下一番景象：



从图中可以看出，构成LevelDb静态结构的包括六个主要部分：内存中的MemTable和Immutable MemTable以及磁盘上的几种主要文件：Current文件，Manifest文件，log文件以及SSTable文件。当然，LevelDb除了这 六个主要部分还有一些辅助的文件，但是以上六个文件和数据结构是LevelDb的主体构成元素。

LevelDb的Log文件和Memtable与Bigtable论文中介绍的是一致的，当应用写入一条Key:Value记录的时候，LevelDb会 先往log文件里写入，成功后将记录插进Memtable中，这样基本就算完成了写入操作，因为一次写入操

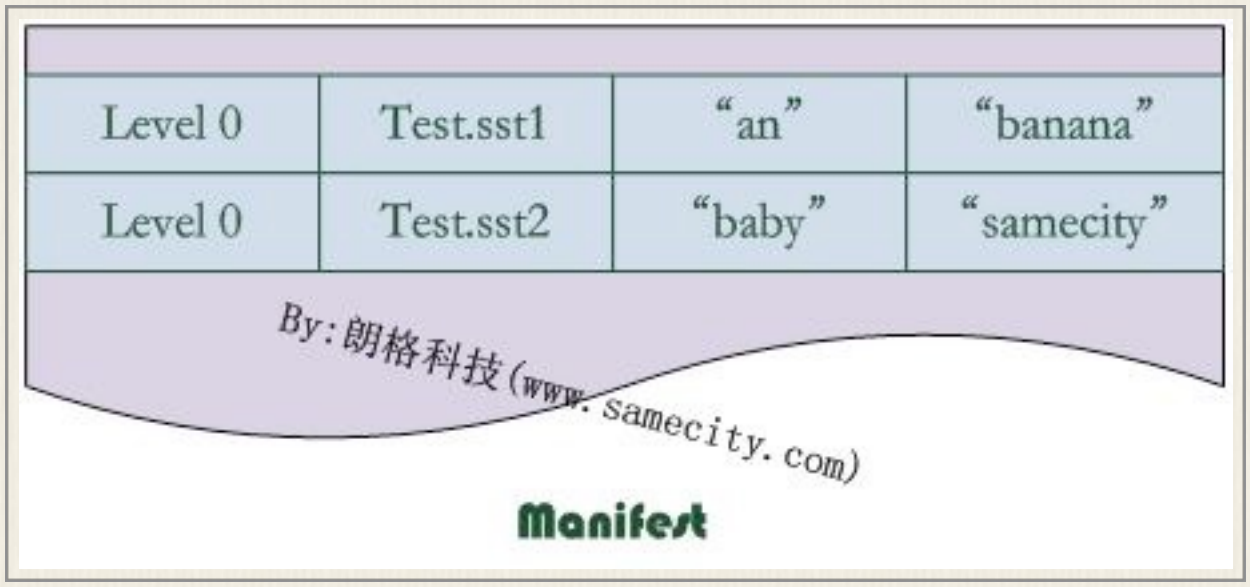
作只涉及一次磁盘顺序写和一次内存写入，所以这 是为何说LevelDb写入速度极快的主要原因。

Log文件在系统中的作用主要是用于系统崩溃恢复而不丢失数据，假如没有Log文件，因为写入的记录刚开始是保存在内存中的，此时如果系统崩溃，内存中的 数据还没有来得及Dump到磁盘，所以会丢失数据（Redis就存在这个问题）。为了避免这种情况，LevelDb在写入内存前先将操作记录到Log文件 中，然后再记入内存中，这样即使系统崩溃，也可以从Log文件中恢复内存中的Memtable，不会造成数据的丢失。

当Memtable插入的数据占用内存到了一个界限后，需要将内存的记录导出到外存文件中，LevleDb会生成新的Log文件和Memtable，原先 的Memtable就成为Immutable Memtable，顾名思义，就是说这个Memtable的内容是不可更改的，只能读不能写入或者删除。新到来的数据被记入新的Log文件和 Memtable，LevelDb后台调度会将Immutable Memtable的数据导出到磁盘，形成一个新的SSTable文件。SSTable就是由内存中的数据不断导出并进行Compaction操作后形成 的，而且SSTable的所有文件是一种层级结构，第一层为Level 0，第二层为Level 1，依次类推，层级逐渐增高，这也是为何称之为LevelDb的原因。

SSTable中的文件是Key有序的，就是说在文件中小key记录排在大Key记录之前，各个Level的SSTable都是如此，但是这里需要注意的 一点是：Level 0的SSTable文件（后缀为.sst）和其它Level的文件相比有特殊性：这个层级内的.sst文件，两个文件可能存在key重叠，比如有两个 level 0的sst文件，文件A和文件B，文件A的key范围是：{bar, car}，文件B的Key范围是{blue,samecity}，那么很可能两个文件都存在key=”blood”的记录。对于其它Level的 SSTable文件来说，则不会出现同一层级内.sst文件的key重叠现象，就是说Level L中任意两个.sst文件，那么可以保证它们的key值是不会重叠的。这点需要特别注意，后面您会看到很多操作的差异都是由于这个原因造成的。

SSTable中的某个文件属于特定层级，而且其存储的记录是key有序的，那么必然有文件中的最小key和最大key，这是非常重要的信 息，LevelDb应该记下这些信息。Manifest就是干这个的，它记载了SSTable各个文件的管理信息，比如属于哪个Level，文件名称叫 啥，最小key和最大key各自是多少。下图是Manifest所存储内容的示意：



图中只显示了两个文件（manifest会记载所有SSTable文件的这些信息），即Level 0的test.sst1和test.sst2文件，同时记载了这些文件各自对应的key范围，比如test.sstt1的key范围是“an”到 “banana”，而文件test.sst2的key范围是“baby”到“samecity”，可以看出两者的key范围是有重叠的。

Current文件是干什么的呢？这个文件的内容只有一个信息，就是记载当前的manifest文件名。因为在LevleDb的运行过程中，随着 Compaction的进行，SSTable文件会发生变化，会有新的文件产生，老的文件被废弃，Manifest也会跟着反映这种变化，此时往往会新生 成Manifest文件来记载这种变化，而Current则用来指出哪个Manifest文件才是我们关心的那个Manifest文件。

以上介绍的内容就构成了LevelDb的整体静态结构，在LevelDb日知录接下来的内容中，朗格科技会首先介绍重要文件或者内存数据的具体数据布局与结构。

LevelDb日知录之三 log文件

上节内容讲到log文件在LevelDb中的主要作用是系统故障恢复时，能够保证不会丢失数据。因为在将记录写入内存的Memtable之前，会先写入Log文件，这样即使系统发生故障，Memtable中的数据没有来得及Dump到

磁盘的SSTable文件，LevelDB也可以根据log文件恢复内 存的Memtable数据结构内容，不会造成系统丢失数据，在这点上LevelDb和Bigtable是一致的。

下面朗格科技带大家看看log文件的具体物理和逻辑布局是怎样的，LevelDb对于一个log文件，会把它切割成以32K为单位的物理Block，每次读取的单位以一个Block作为基本读取单位，下图展示的log文件由3个Block构成，所以从物理布局来讲，一个log文件就是由连续的32K大小 Block构成的。

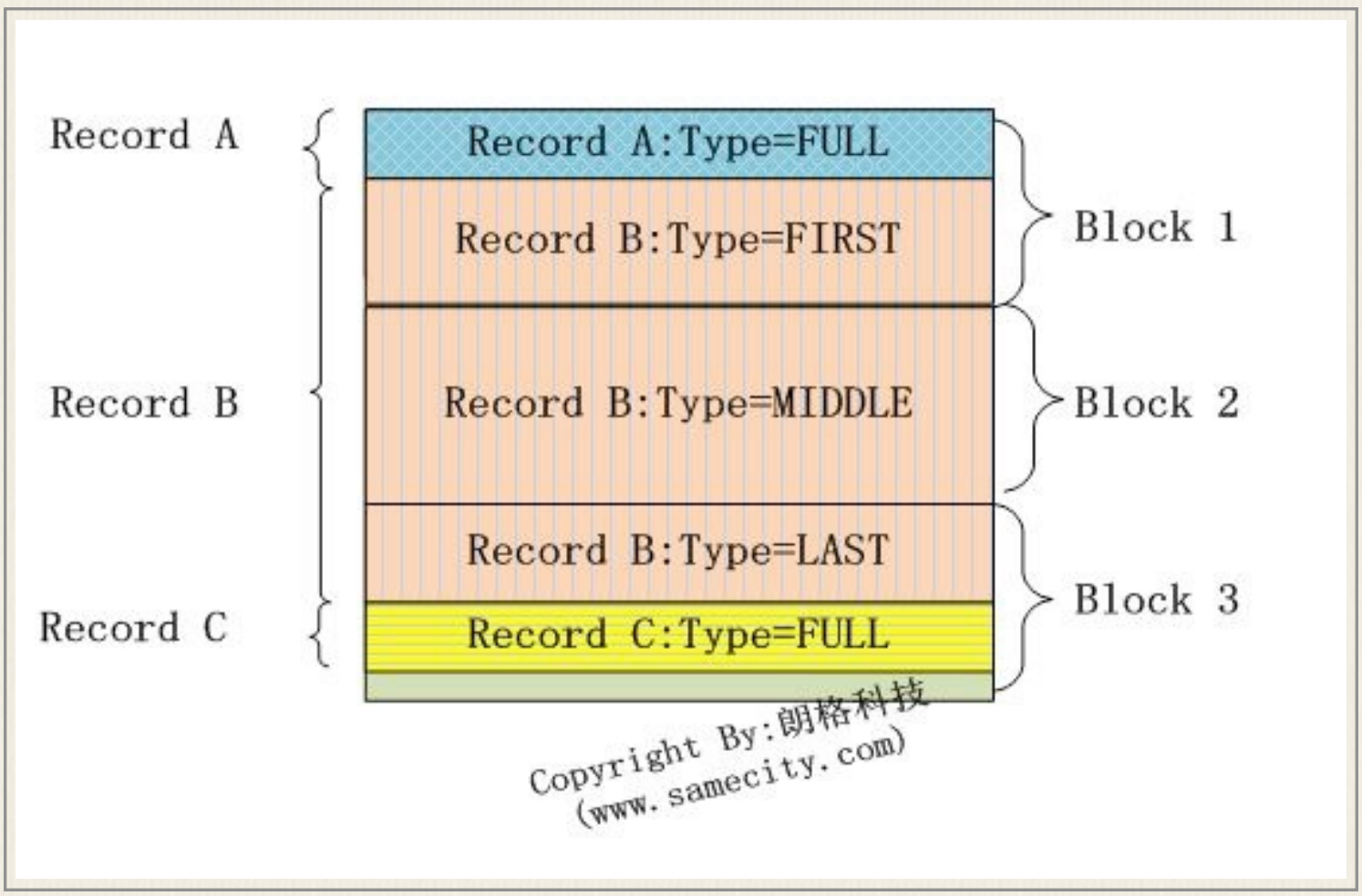


图3.1 log文件布局

在应用的视野里是看不到这些Block的，应用看到的是一系列的Key:Value对，在LevelDb内部，会将一个Key:Value对看做一条记录 的数据，另外在这个数据前增加一个记录头，用来记载一些管理信息，以方便内部处理，图3.2显示了一个记录在LevelDb内部是如何表示的。

图3.2 记录结构

记录头包含三个字段，ChechSum是对“类型”和“数据”字段的校验码，为了避免处理不完整或者是被破坏的数据，当 LevelDb读取记录数据时候会对数据进行校验，如果发现和存储的CheckSum相同，说明数据完整无破坏，可以继续后续流程。“记录长度”记载了数 据的大小，“数据”则是上面讲的Key:Value数值对，“类型”字段则指出了每条记录的逻辑结构和log文件物理分块结构之间的关系，具体而言，主要 有以下四种类型：FULL/FIRST/MIDDLE/LAST。

如果记录类型是FULL，代表了当前记录内容完整地存储在一个物理Block里，没有被不同的物理Block切割开；如果记录被相邻的物理Block切割开，则类型会是其他三种类型中的一种。我们以图3.1所示的例子来具体说明。

假设目前存在三条记录，Record A，Record B和Record C，其中Record A大小为10K，Record B 大小为80K，Record C大小为12K，那么其在log文件中的逻辑布局会如图3.1所示。Record A是图中蓝色区域所示，因为大小为10K<32K，能够放在一个物理Block中，所以其类型为FULL；Record B 大小为80K，而Block 1因为放入了Record A，所以还剩下22K，不足以放下Record B，所以在Block 1的剩余部分放入Record B的开头一部分，类型标识为FIRST，代表了是一个记录的起始部分；Record B还有58K没有存储，这些只能依次放在后续的物理Block里面，因为Block 2大小只有32K，仍然放不下Record B的剩余部分，所以Block 2全部用来放Record B，且标识类型为MIDDLE，意思是这是Record B中间一段数据；Record B剩下的部分可以完全放在Block 3中，类型标识为LAST，代表了这是Record B的末尾数据；图中黄色的Record C因为大小为12K，Block 3剩下的空间足以全部放下它，所以其类型标识为FULL。

从这个小例子可以看出逻辑记录和物理Block之间的关系，LevelDb一次物理读取为一个Block，然后根据类型情况拼接出逻辑记录，供后续流程处理。

LevelDb日知录之四 SSTable文件

SSTable是Bigtable中至关重要的一块，对于LevelDb来说也是如此，对LevelDb的SSTable实现细节的了解也有助于了解Bigtable中一些实现细节。

本节内容主要讲述SSTable的静态布局结构，朗格科技曾在“LevelDb日知录之二：整体架构”中说过，SSTable文件形成了不同Level的 层级结构，至于这个层级结构是如何形成的我们放在后面Compaction一节细说。本节主要介绍SSTable某个文件的物理布局和逻辑布局结构，这对 了解LevelDb的运行过程很有帮助。

LevelDb不同层级有很多SSTable文件（以后缀.sst为特征），所有.sst文件内部布局都是一样的。上节介绍Log文件是物理分块 的，SSTable也一样会将文件划分为固定大小的物理存储块，但是两者逻辑布局大不相同，根本原因是：Log文件中的记录是Key无序的，即先后记录的 key大小没有明确大小关系，而.sst文件内部则是根据记录的Key由小到大排列的，从下面介绍的SSTable布局可以体会到Key有序是为何如此设 计.sst文件结构的关键。

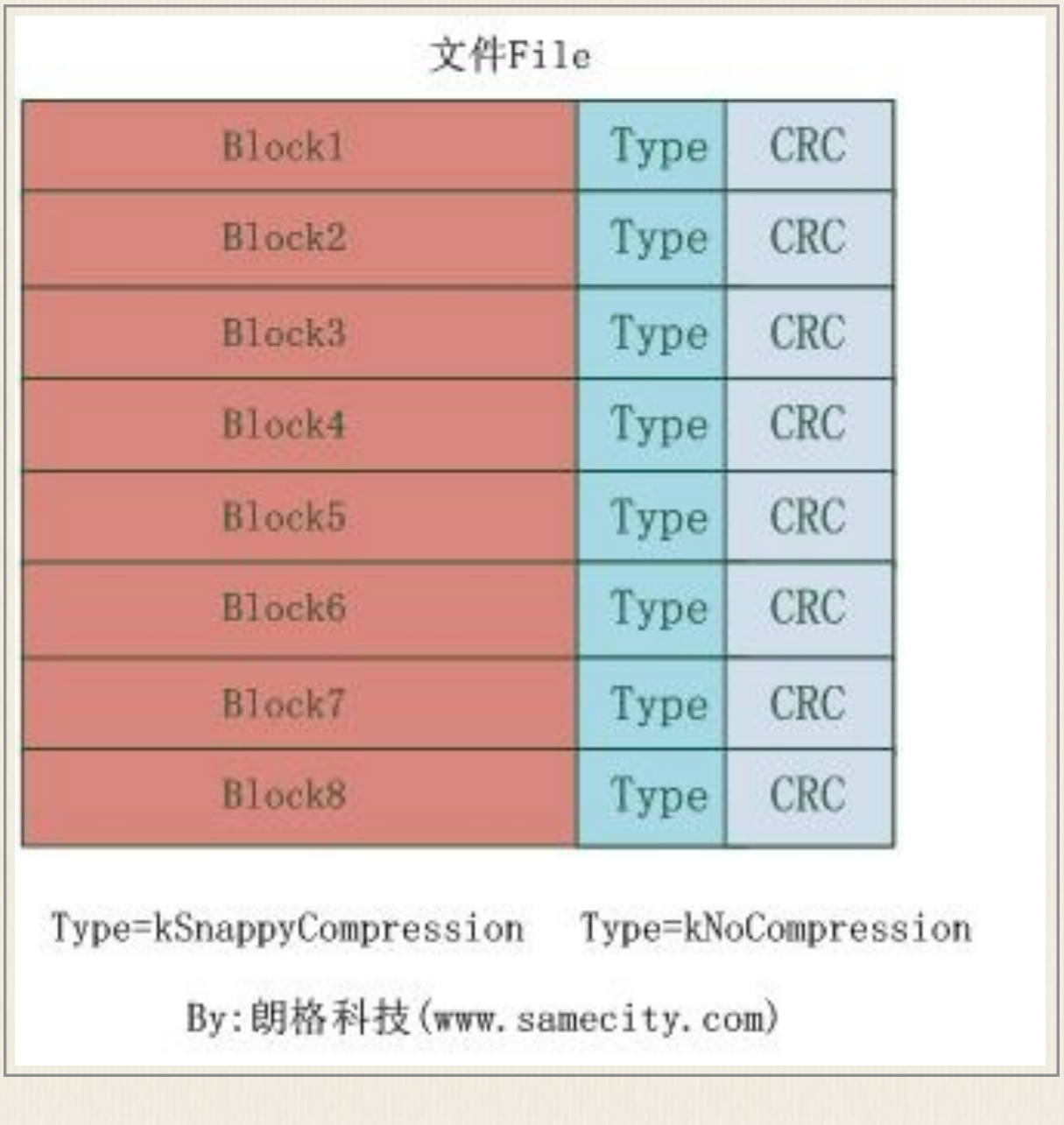


图4.1 .sst文件的分块结构

图4.1展示了一个.sst文件的物理划分结构，同Log文件一样，也是划分为固定大小的存储块，每个Block分为三个部分，红色部分是数据存储 区，蓝色的Type区用于标识数据存储区是否采用了数据压缩算法（Snappy压缩或者无压缩两种），CRC部分则是数据校验码，用于判别数据是否在生成 和传输中出错。

以上是.sst的物理布局，下面介绍.sst文件的逻辑布局，所谓逻辑布局，就是说尽管大家都是物理块，但是每一块存储什么内容，内部又有什么结构等。图4.2展示了.sst文件的内部逻辑解释。

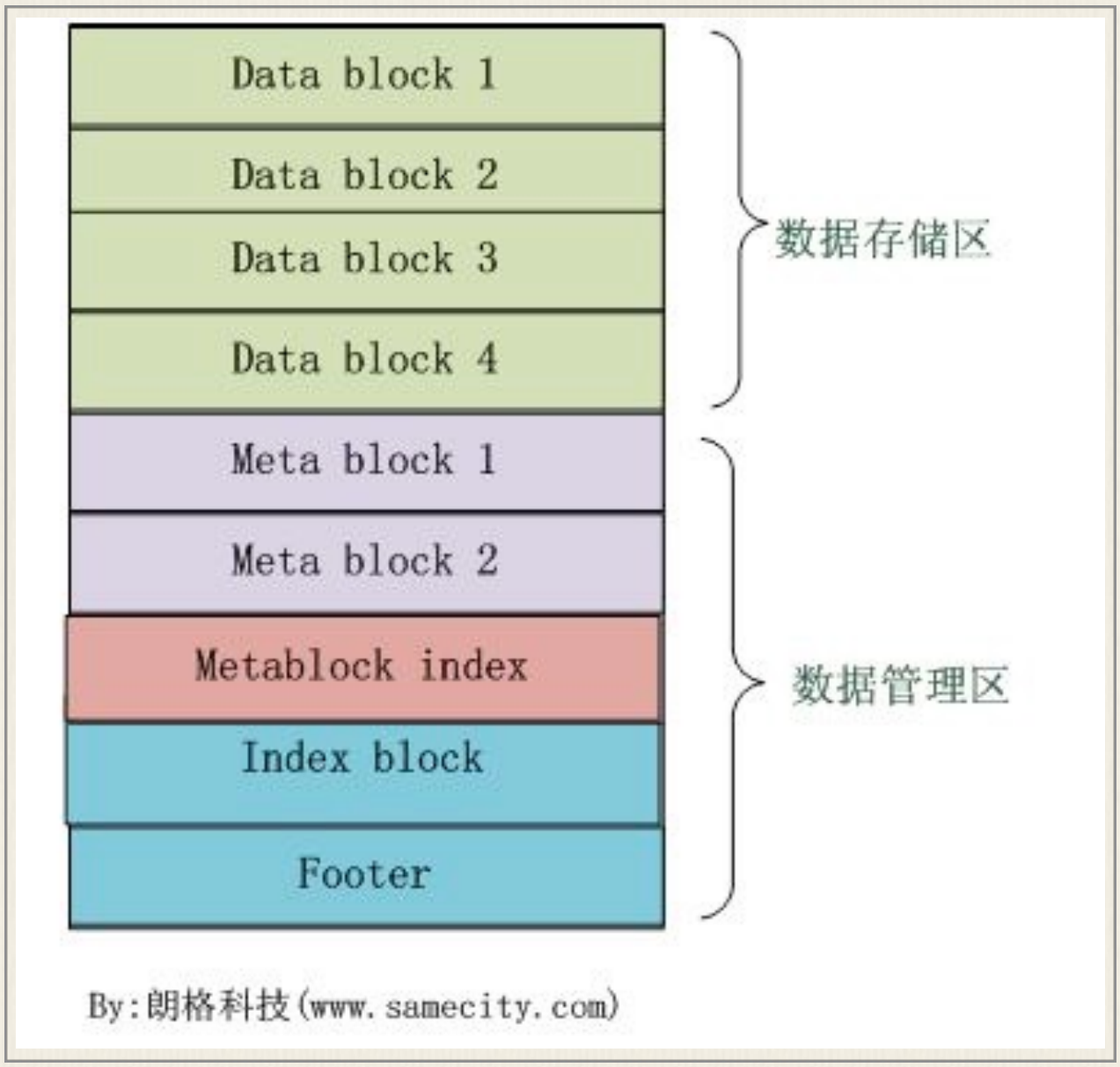


图4.2 逻辑布局

从图4.2可以看出，从大的方面，可以将.sst文件划分为数据存储区和数据管理区，数据存储区存放实际的Key:Value数据，数据管理区则提供一些索引指针等管理数据，目的是更快速便捷的查找相应的记录。两个区域都是在上述的分块基础上的，就是说文件的前面若干块实际存储KV数据，后面数据管理区存 储管理数据。管理数据又分为四种不同类型：紫色的Meta Block，红色的MetaBlock 索引和蓝色的数据索引块以及一个文件尾部块。

LevelDb 1.2版对于Meta Block尚无实际使用，只是保留了一个接口，估计会在后续版本中加入内容，下面我们看看数据索引区和文件尾部Footer的内部结构。

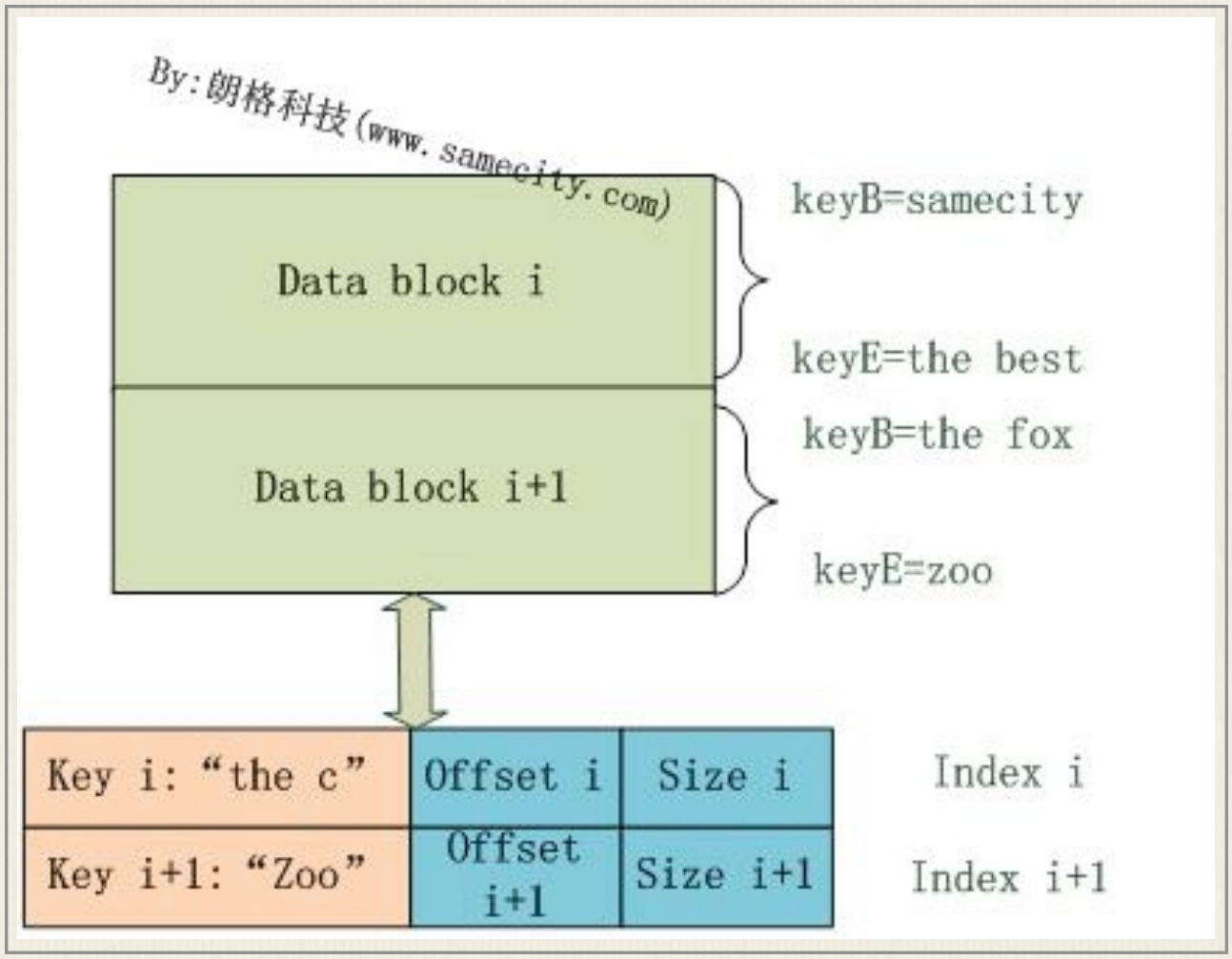


图4.3 数据索引

图4.3是数据索引的内部结构示意图。再次强调一下，Data Block内的KV记录是按照Key由小到大排列的，数据索引区的每条记录是对某个Data Block建立的索引信息，每条索引信息包含三个内容，以图4.3所示的数据块i的索引Index i来说：红色部分的第一个字段记载大于等于数据块i中最大的Key值的那个Key，第二个字段指出数据块i在.sst文件中的起始位置，第三个字段指出 Data Block i的大小（有时候是有数据压缩的）。后面两个字段好理解，是用于定位数据块在文件中的位置的，第一个字段需要详细解释一下，在索引里保存的这个Key值未 必一定是某条记录的Key,以图4.3的例子来说，假设数据块i 的最小Key=“samecity”，最大Key=“the best”;数据块i+1的最小Key=“the fox”,最大Key=“zoo”,那么对于数据块i的索引Index i来说，其第一个字段记载大于等于数据块i的最大Key(“the best”)同时要小于数据块i+1的最小Key(“the fox”)，所以例子中Index i的第一个字段是：“the c”，这个是满足要求的；而Index i+1的第一个字段则是“zoo”，即数据块i+1的最大Key。

文件末尾Footer块的内部结构见图4.4，metaindex_handle指出了metaindex block的起始位置和大小；inex_handle指出了index Block的起始地址和大小；这两个字段可以理解为索引的索引，是为了正确读出索引值而设立的，后面跟着一个填充区和魔数。

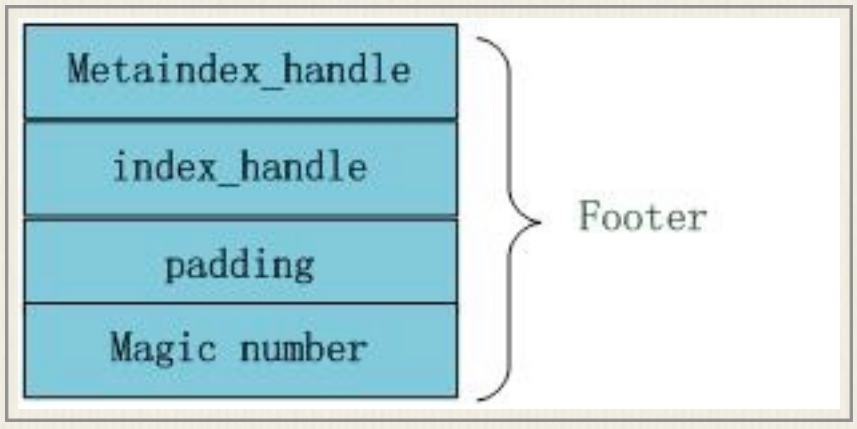


图4.4 Footer

上面主要介绍的是数据管理区的内部结构，下面我们看看数据区的一个Block的数据部分内部是如何布局的（图4.1中的红色部分），图4.5是其内部布局示意图。

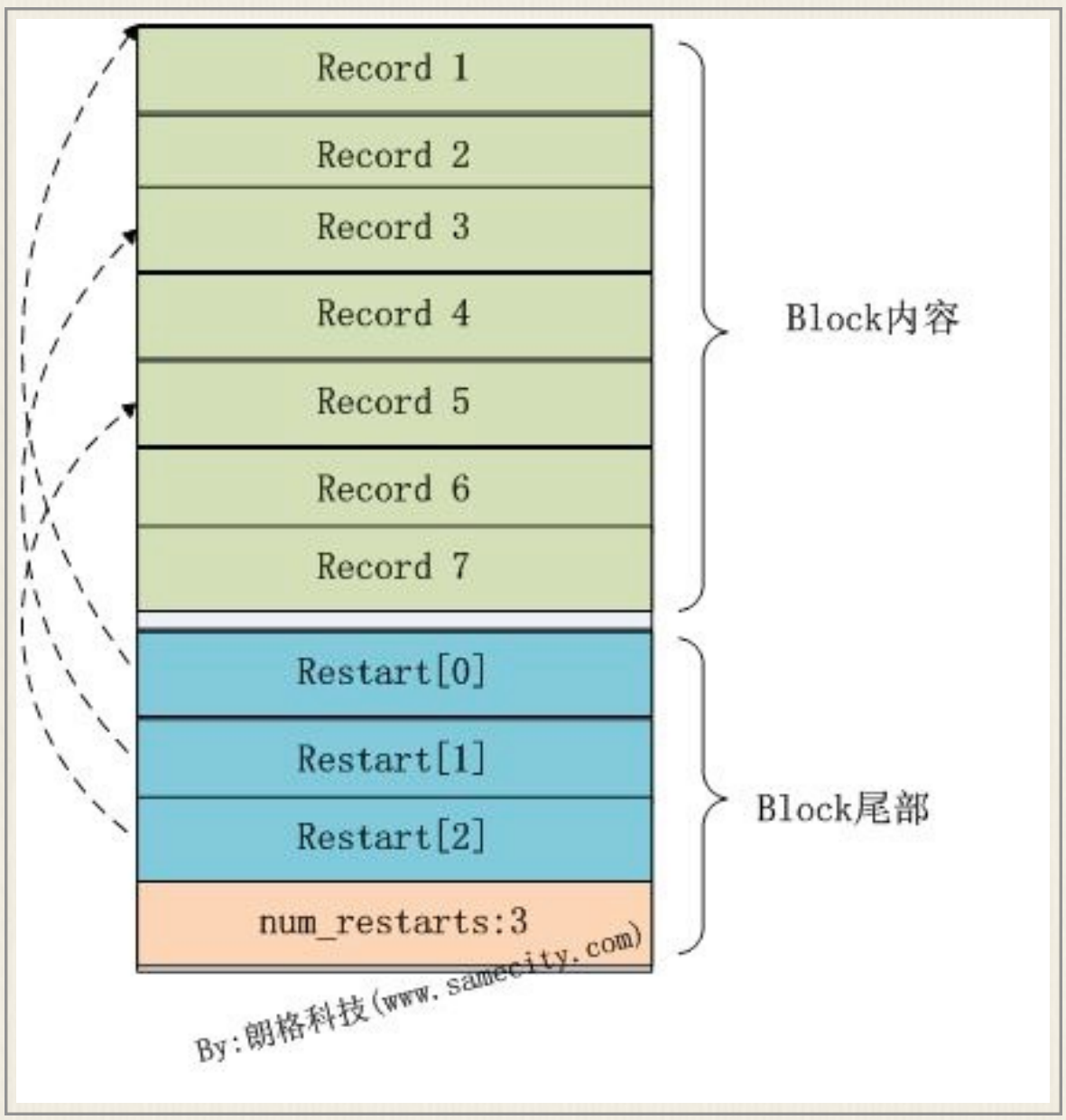


图4.5 数据Block内部结构

从图中可以看出，其内部也分为两个部分，前面是一个个KV记录，其顺序是根据Key值由小到大排列的，在Block尾部则是一些“重启点”（Restart Point）,其实是一些指针，指出Block内容中的一些记录位置。

“重启点”是干什么的呢？我们一再强调，Block内容里的KV记录是按照Key大小有序的，这样的话，相邻的两条记录很可能Key部分存在重叠，比如 key i=“the Car”，Key i+1=“the color”,那么两者存在重叠部分“the c”，为了减少Key的存储量，Key i+1可以只存储和上一条Key不同的部分“olor”，两者的共同部分从Key i中可以获得。记录的Key在Block内容部分就是这么存储的，主要目的是减少存储开销。“重启点”的意思是：在这条记录开始，不再采取只记载不同的 Key部分，而是重新记录所有的Key值，假设Key i+1是一个重启点，那么Key里面会完整存储“the color”，而不是采用简略的“olor”方式。Block尾部就是指出哪些记录是这些重启点的。

图4.6 记录格式

在Block内容区，每个KV记录的内部结构是怎样的？图4.6给出了其详细结构，每个记录包含5个字段：key共享长度，比如上面的“olor”记录， 其key和上一条记录共享的Key部分长度是“the c”的长度，即5；key非共享长度，对于“olor”来说，是4；value长度指出Key:Value中Value的长度，在后面的Value内容字 段中存储实际的Value值；而key非共享内容则实际存储“olor”这个Key字符串。

上面讲的这些就是.sst文件的全部内部奥秘。

LevelDb日知录之五 MemTable

LevelDb日知录前述小节大致讲述了磁盘文件相关的重要静态结构，本小节讲述内存中的数据结构Memtable，Memtable在整个体系 中的重要地位也不言而喻。总体而言，所有KV数据都是存储在Memtable，Immutable Memtable和SSTable中的，Immutable Memtable从结构上讲和Memtable是完全一样的，区别仅仅在于其是只读的，不允许写入操作，而Memtable则是允许写入和读取的。当 Memtable写入的数据占用内存到达指定数量，则自动转换为Immutable Memtable，等待Dump到磁盘中，系统会自动生成新的Memtable供写操作写入新数据，理解了Memtable，那么Immutable Memtable自然不在话下。

LevelDb的MemTable提供了将KV数据写入，删除以及读取KV记录的操作接口，但是事实上Memtable并不存在真正的删除操作,删除某 个Key的Value在Memtable内是作为插入一条记录实施的，但是会打上一个Key的删除标记，真正的删除操作是Lazy的，会在以后的 Compaction过程中去掉这个KV。

需要注意的是，LevelDb的Memtable中KV对是根据Key大小有序存储的，在系统插入新的KV时，LevelDb要把这个KV插到合适的 位置上以保持这种Key有序性。其实，LevelDb的Memtable类只是一个接口类，真正的操作是通过背后的SkipList来做的，包括插入操作 和读取操作等，所以Memtable的核心数据结构是一个SkipList。

SkipList是由William Pugh发明。他在Communications of the ACM June 1990, 33(6) 668-676 发表了Skip lists: a probabilistic alternative to balanced trees，在该论文中详细解释了SkipList的数据结构和插入删除操作。

SkipList是平衡树的一种替代数据结构，但是和红黑树不相同的是，SkipList对于树的平衡的实现是基于一种随机化的算法的，这样也就是说SkipList的插入和删除的工作是比较简单的。

关于SkipList的详细介绍可以参考这篇文章：http://www.cnblogs.com/xuqiang/archive/2011/05/22/2053516.html，讲述的很清楚，LevelDb的SkipList基本上是一个具体实现，并无特殊之处。

SkipList不仅是维护有序数据的一个简单实现，而且相比较平衡树来说，在插入数据的时候可以避免频繁的树节点调整操作，所以写入效率是很高的，LevelDb整体而言是个高写入系统，SkipList在其中应该也起到了很重要的作用。Redis为了加快插入操作，也使用了SkipList来 作为内部实现数据结构。

LevelDB性能分析和表现(原文链接：http://blog.yufeng.info/archives/1327)

Leveldb是一个google实现的非常高效的kv数据库，目前的版本1.2能够支持billion级别的数据量了。 在这个数量级别下还有着非常高的性能，主要归功于它的良好的设计。特别是LSM算法。

那么数据库最怕的的随机IO他是如何解决的呢?

先说随机写，它的写都是先记录到日志文件去的，在日志文件满之前只是简单的更新memtable,那么就把随机写转化成了顺序写。在日志满了后，把 日志里面的数据排序写成sst表同时和之前的sst进行合并，这个动作也是顺序读和写。大家都知道传统磁盘raid的顺序读写吞吐量是很大的，100M左 右是没有问题。在写日志文件的时候，用到是buffer IO，也就是说如果操作系统有足够的内存，这个读写全部由操作系统缓冲，效果非常好。即使是sync写模式，也是以数据累计到4K为一个单位写的，所以效 率高。

那么随机读呢?这个它解决不了。但是ssd盘最擅长随机读了。这个硬件很自然的解决了这个问题。

所以leveldb的绝配是ssd盘的raid.

leveldb标准版本编译见浅谈LevelDB在ubuntu 11.04下编译失败的问题，由于标准版本用到了c++ 0x的特性，在RHEL平台下没得到支持，所以为了移植性, basho为它做了标准c++版本的port, 见目录c_src/leveldb. 他之所以用c++ 0x标准主要是用到里面的原子库，basho的port用了libatomicops搞定这个问题.

我们的测试采用的就是这个版本, 我们分别测试了1000万， 1亿，10亿数据量下的leveldb表现，发现随着数据集的变化性能变化不大。

由于leveldb默认的sst文件是2M, 在数据集达到100G的时候要占用几万个文件，我修改了：

1. version_set.cc：

```
23 static const int kTargetFileSize = 32 * 1048576;
```

让默认的文件变成32M,减少目录的压力。

我的测试环境是:

1. $uname -r

2. 2.6.18-164.el5 #RHEL 5U4

3. # 10* SAS 300G raid卡，fusionIO 320G, Flashcache,内存96G，24 * Intel(R) Xeon(R) CPU

top说:

1.

```
21782 root      18   0 1273m 1.1g 2012 R 85.3  1.2   1152:34 db_bench
```

iostat说:

1. $iostat -dx 5

2. ...

3.

```
sdb1              0.40     0.00  3.40  0.00    30.40     0.00     8.94     0.02    4.65
  4.65   1.58
```

4.

```
fioa              0.00     0.00 2074.80  3.80 16598.40    30.40     8.00     0.00
0.13   0.00   0.00
```

5.
dm-0 0.00 0.00 1600.00 0.00 16630.40 0.00 10.39 0.25
0.15 0.15 24.76

6. ...

该测试中请注意snap-
py压缩没有打开，如果有压缩性能还会高很多，因为IO少了一半。

write_buffer_size=$((256*1024*1024))，log大小设成256M，这样减少切换日志的开销和减少数据合并的频率。

同时应该注意到db_bench是单线程程序，还有一个compact线程，所以最多的时候这个程序只能跑到200%的cpu, IO util也不是很高. 换句话说如果是多线程程序的话性能还要N倍的提高。

我们来看下实际的性能数字:

1. #1千万条记录
2.
$sudo ./db_bench --num=10000000 --write_buffer_size=$((256*1024*102
4))
3. LevelDB: version 1.2
4. Date: Fri May 27 17:14:33 2011
5. CPU: 24 * Intel(R) Xeon(R) CPU X5670 @ 2.93GHz
6. CPUCache: 12288 KB
7. Keys: 16 bytes each
8. Values: 100 bytes each (50 bytes after compression)
9. Entries: 10000000
10. RawSize: 1106.3 MB (estimated)
11. FileSize: 629.4 MB (estimated)
12. write_buffer_size=268435456

```
WARNING: Snappy compression is not enabled
------------------------------------------------
fillseq      :       2.134 micros/op;   51.8 MB/s
fillsync     :      70.722 micros/op;    1.6 MB/s (100000 ops)
fillrandom   :       5.229 micros/op;   21.2 MB/s
overwrite    :       5.396 micros/op;   20.5 MB/s
readrandom   :      65.729 micros/op;
readrandom   :      43.086 micros/op;
readseq      :       0.882 micros/op;  125.4 MB/s
readreverse  :       1.200 micros/op;   92.2 MB/s
compact      : 24599514.008 micros/op;
readrandom   :      12.663 micros/op;
readseq      :       0.372 micros/op;  297.4 MB/s
readreverse  :       0.559 micros/op;  198.0 MB/s
fill100K     :     349.894 micros/op;  272.6 MB/s (10000 ops)
crc32c       :       4.759 micros/op;  820.8 MB/s (4K per op)
snappycomp   :       3.099 micros/op; (snappy failure)
snappyuncomp :       2.146 micros/op; (snappy failure)

#1亿条记录

$sudo ./db_bench --num=100000000 --write_buffer_size=$((256*1024*1024))
LevelDB:    version 1.2
Date:       Fri May 27 17:39:19 2011
```

```
CPU:        24 * Intel(R) Xeon(R) CPU           X5670  @ 2.93GHz
CPUCache:   12288 KB
Keys:       16 bytes each
Values:     100 bytes each (50 bytes after compression)
Entries:    100000000
RawSize:    11062.6 MB (estimated)
FileSize:   6294.3 MB (estimated)
write_buffer_size=268435456
WARNING: Snappy compression is not enabled
------------------------------------------------
fillseq      :       2.140 micros/op;   51.7 MB/s
fillsync     :      70.592 micros/op;    1.6 MB/s (1000000 ops)
fillrandom   :       6.033 micros/op;   18.3 MB/s
overwrite    :       7.653 micros/op;   14.5 MB/s
readrandom   :      44.833 micros/op;
readrandom   :      43.963 micros/op;
readseq      :       0.561 micros/op;  197.1 MB/s
readreverse  :       0.809 micros/op;  136.8 MB/s
compact      : 123458261.013 micros/op;
readrandom   :      14.079 micros/op;
readseq      :       0.378 micros/op;  292.5 MB/s
readreverse  :       0.567 micros/op;  195.2 MB/s
fill100K     :    1516.707 micros/op;   62.9 MB/s (100000 ops)
crc32c       :       4.726 micros/op;  826.6 MB/s (4K per op)
```

```
60. snappycomp   :       1.907 micros/op; (snappy failure)
61. snappyuncomp :       0.954 micros/op; (snappy failure)
62.
63. #10亿条记录
64.
$sudo ./db_bench --num=1000000000 --write_buffer_size=$((256*1024*1
024))
65. Password:
66. LevelDB:    version 1.2
67. Date:       Sun May 29 17:04:14 2011
68. CPU:        24 * Intel(R) Xeon(R) CPU           X5670  @ 2.93GHz
69. CPUCache:   12288 KB
70. Keys:       16 bytes each
71. Values:     100 bytes each (50 bytes after compression)
72. Entries:    1000000000
73. RawSize:    110626.2 MB (estimated)
74. FileSize:   62942.5 MB (estimated)
75. write_buffer_size=268435456
76. WARNING: Snappy compression is not enabled
77. ------------------------------------------------
78. fillseq      :       2.126 micros/op;   52.0 MB/s
79. fillsync     :      63.644 micros/op;    1.7 MB/s (10000000 ops)
80. fillrandom   :      10.267 micros/op;   10.8 MB/s
81. overwrite    :      14.339 micros/op;    7.7 MB/s
82. ...比较慢待补充
```

总结: Leveldb 是个很好的kv库，重点解决了随机IO性能不好的问题，多线程更新的性能非常好.

LevelDb日知录之六:写入与删除记录

在之前的五节LevelDb日知录中，朗格科技介绍了LevelDb的一些静态文件及其详细布局，从本节开始，我们看看LevelDb的一些动态操作，比如读写记录，Compaction，错误恢复等操作。

本节介绍levelDb的记录更新操作，即插入一条KV记录或者删除一条KV记录。levelDb的更新操作速度是非常快的，源于其内部机制决定了这种更新操作的简单性。

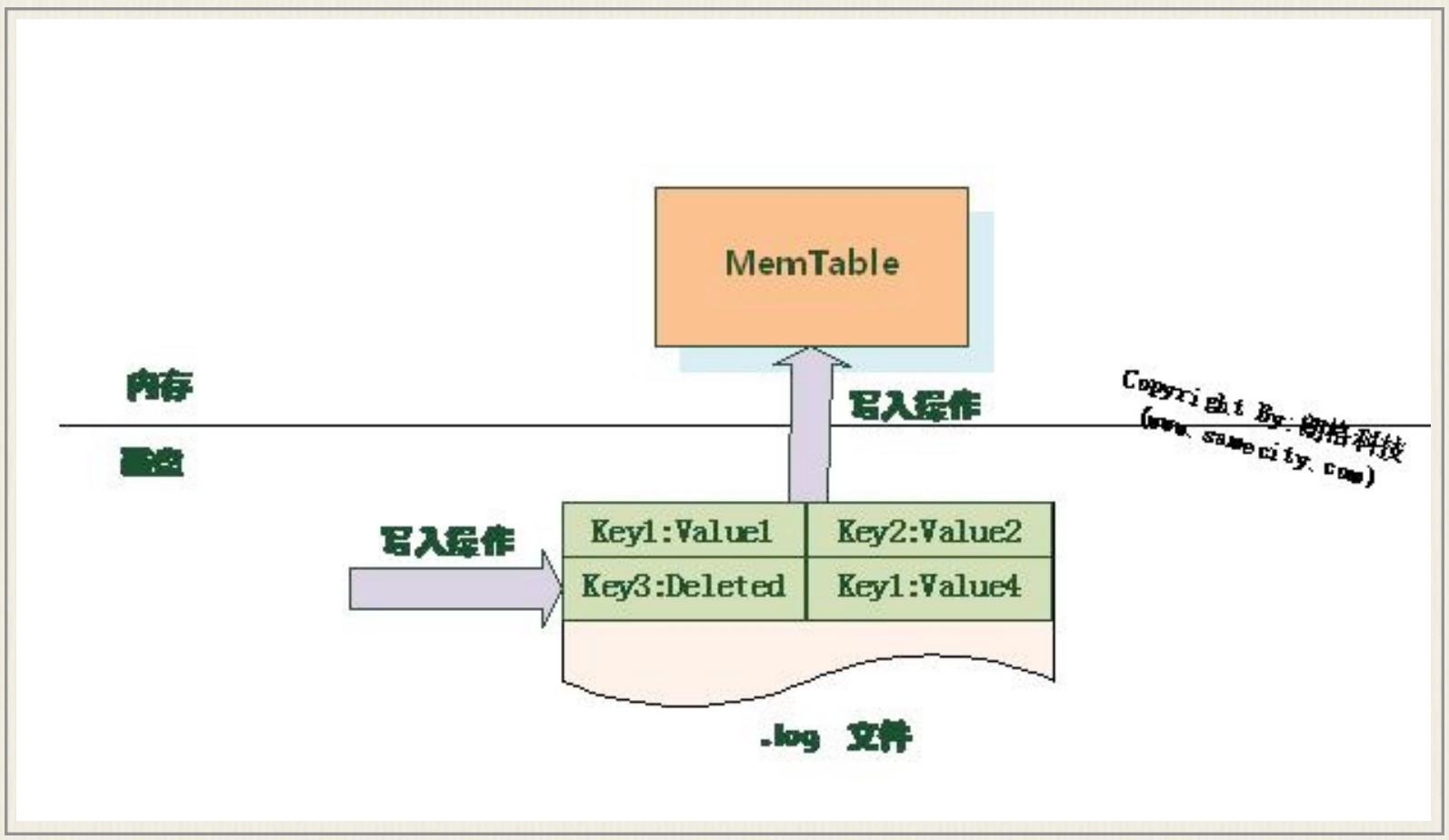


图6.1 LevelDb写入记录

图6.1是levelDb如何更新KV数据的示意图，从图中可以看出，对于一个插入操作Put(Key,Value)来说，完成插入操作包含两个具 体步骤：首先是将这条KV记录以顺序写的方式追加到之前介绍过的log文件末尾，因为尽管这是一个磁盘读写操作，但是文件的顺序追加写入效率是很高的，所 以并不

会导致写入速度的降低；第二个步骤是:如果写入log文件成功，那么将这条KV记录插入内存中的Memtable中，前面介绍过，Memtable 只是一层封装，其内部其实是一个Key有序的SkipList列表，插入一条新记录的过程也很简单，即先查找合适的插入位置，然后修改相应的链接指针将新 记录插入即可。完成这一步，写入记录就算完成了，所以一个插入记录操作涉及一次磁盘文件追加写和内存SkipList插入操作，这是为何levelDb写 入速度如此高效的根本原因。

从上面的介绍过程中也可以看出：log文件内是key无序的，而Memtable中是key有序的。

那么如果是删除一条KV记录呢？对于levelDb来说，并不存在立即删除的操作，而是与插入操作相同的，区别是，插入操作插入的是Key:Value 值，而删除操作插入的是“Key:删除标记”，并不真正去删除记录，而是后台Compaction的时候才去做真正的删除操作。

levelDb 的写入操作就是如此简单。真正的麻烦在后面将要介绍的读取操作中。

LevelDb日知录之七如何根据Key读取记录？

LevelDb是针对大规模Key/Value数据的单机存储库，从应用的角度来看，LevelDb就是一个存储工具。而作为称职的存储工具，常见的调用 接口无非是新增KV，删除KV，读取KV，更新Key对应的Value值这么几种操作。LevelDb的接口没有直接支持更新操作的接口，如果需要更新某 个Key的Value,你可以选择直接生猛地插入新的KV，保持Key相同，这样系统内的key对应的value就会被更新；或者你可以先删除旧的KV， 之后再插入新的KV，这样比较委婉地完成KV的更新操作。

假设应用提交一个Key值，下面我们看看LevelDb是如何从存储的数据中读出其对应的Value值的。图7-1是LevelDb读取过程的整体示意图。

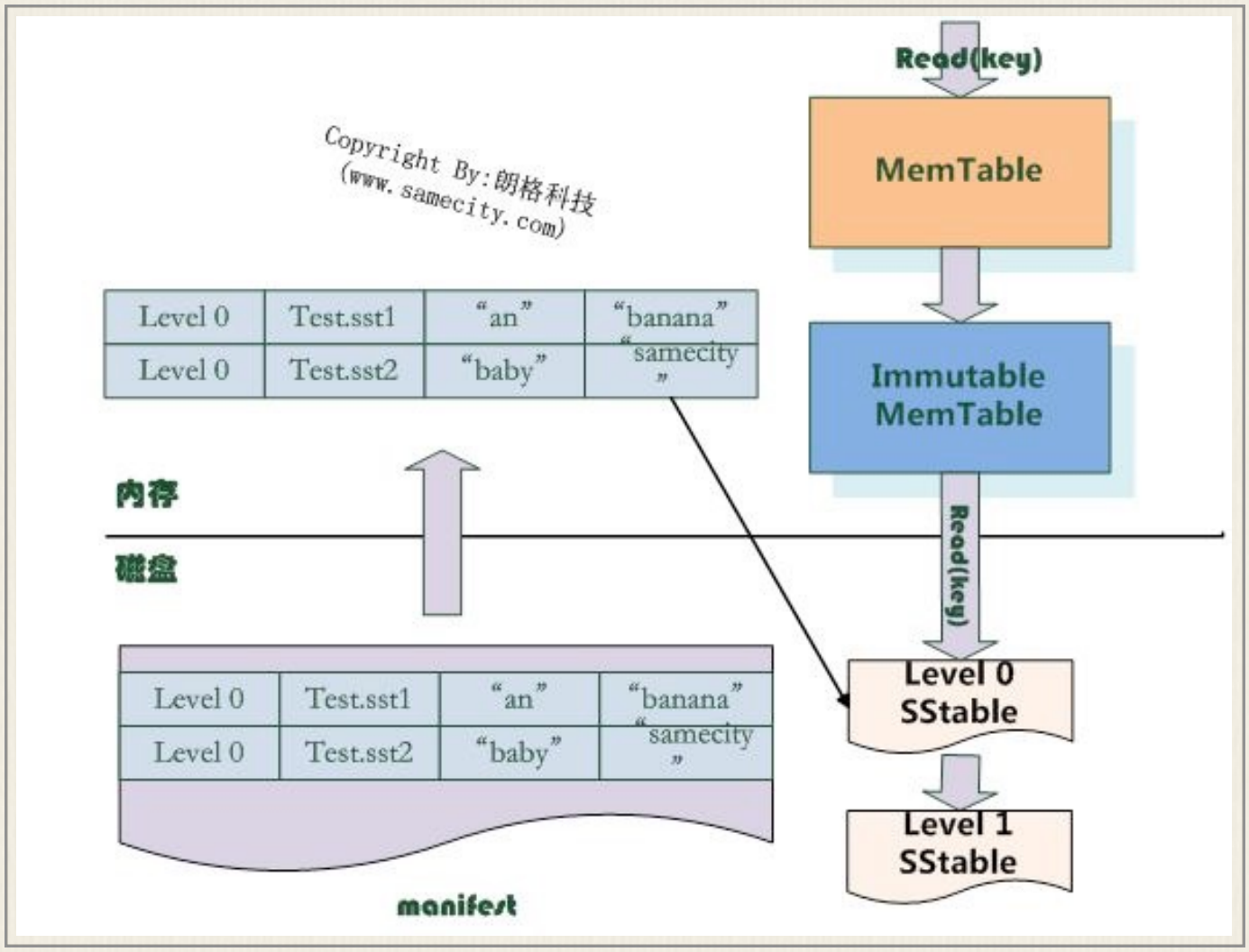


图7-1 LevelDb读取记录流程

LevelDb首先会去查看内存中的Memtable，如果Memtable中包含key及其对应的value，则返回value值即可；如果在 Memtable没有读到key，则接下来到同样处于内存中的Immutable Memtable中去读取，类似地，如果读到就返回，若是没有读到,那么只能万般无奈下从磁盘中的大量SSTable文件中查找。因为SSTable数量 较多，而且分成多个Level，所以在SSTable中读数据是相当蜿蜒曲折的一段旅程。总的读取原则是这样的：首先从属于level 0的文件中查找，如果找到则返回对应的value值，如果没有找到那么到level 1中的文件中去找，如此循环往复，直到在某层SSTable文件中找到这个key对应的value为止（或者查到最高level，查找失败，说明整个系统 中不存在这个Key)。

那么为什么是从Memtable到Immutable Memtable，再从Immutable Memtable到文件，而文件中为何是从低level到高level这么一个查询路径呢？道理何在？之所以选择这么个查询路径，是因为从信息的更新时间 来说，很明显Memtable存储的是最新鲜的KV对；Immutable Memtable中存储的KV数据对的新鲜程度次之；而所有SSTable文件中的KV数据新鲜程度

一定不如内存中的Memtable和 Immutable Memtable的。对于SSTable文件来说，如果同时在level L和Level L+1找到同一个key，level L的信息一定比level L+1的要新。也就是说，上面列出的查找路径就是按照数据新鲜程度排列出来的，越新鲜的越先查找。

为啥要优先查找新鲜的数据呢？这个道理不言而喻，举个例子。比如我们先往levelDb里面插入一条数据 {key="www.samecity.com" value="朗格科技"},过了几天，samecity网站改名为：69同城，此时我们插入数据{key="www.samecity.com" value="69同城"}，同样的key,不同的value；逻辑上理解好像levelDb中只有一个存储记录，即第二个记录，但是在levelDb中 很可能存在两条记录，即上面的两个记录都在levelDb中存储了，此时如果用户查询key="www.samecity.com",我们当然希望找到最 新的更新记录，也就是第二个记录返回，这就是为何要优先查找新鲜数据的原因。

前文有讲：对于SSTable文件来说，如果同时在level L和Level L+1找到同一个key，level L的信息一定比level L+1的要新。这是一个结论，理论上需要一个证明过程，否则会招致如下的问题：为神马呢？从道理上讲呢，很明白：因为Level L+1的数据不是从石头缝里蹦出来的，也不是做梦梦到的，那它是从哪里来的？Level L+1的数据是从Level L 经过Compaction后得到的（如果您不知道什么是Compaction，那么.....也许以后会知道的），也就是说，您看到的现在的 Level L+1层的SSTable数据是从原来的Level L中来的，现在的Level L比原来的Level L数据要新鲜，所以可证，现在的Level L比现在的Level L+1的数据要新鲜。

证毕。

如果您没看明白上面的证明过程，那么请记得往您的IQ卡内充值。

SSTable文件很多，如何快速地找到key对应的value值？在LevelDb中，level 0一直都爱搞特殊化，在level 0和其它level中查找某个key的过程是不一样的。因为level 0下的不同文件可能key的范围有重叠，某个要查询的key有可能多个文件都包含，这样的话LevelDb的策略是先找出level 0中哪些文件包含这个key（manifest文件中记载了level和对应的文件及文件里key的范围信息，LevelDb在内存中保留这种映射表）， 之后按照文件的新鲜程度排序，新的文件排在前面，之后依次查找，读出key对应的

value。而如果是非level 0的话，因为这个level的文件之间key是不重叠的，所以只从一个文件就可以找到key对应的value。

最后一个问题,如果给定一个要查询的key和某个key range包含这个key的SSTable文件，那么levelDb是如何进行具体查找过程的呢？levelDb一般会先在内存中的Cache中查找是否 包含这个文件的缓存记录，如果包含，则从缓存中读取；如果不包含，则打开SSTable文件，同时将这个文件的索引部分加载到内存中并放入Cache中。 这样Cache里面就有了这个SSTable的缓存项，但是只有索引部分在内存中，之后levelDb根据索引可以定位到哪个内容Block会包含这条 key，从文件中读出这个Block的内容，在根据记录一一比较，如果找到则返回结果，如果没有找到，那么说明这个level的SSTable文件并不包 含这个key，所以到下一级别的SSTable中去查找。

从之前介绍的LevelDb的写操作和这里介绍的读操作可以看出，相对写操作，读操作处理起来要复杂很多，所以写的速度必然要远远高于读数据的速度，也就 是说，LevelDb比较适合写操作多于读操作的应用场合。而如果应用是很多读操作类型的，那么顺序读取效率会比较高，因为这样大部分内容都会在缓存中找 到，尽可能避免大量的随机读取操作。

LevelDb日知录之八:Compaction

前文有述，对于LevelDb来说，写入记录操作很简单，删除记录仅仅写入一个删除标记就算完事了，但是读取记录比较复杂，需要在内存以及各个层级 文件中依照新鲜程度依次查找，代价很高。为了加快读取速度，levelDb采取了compaction的方式来对已有的记录进行整理压缩，通过这种方式，来删除掉一些不再有效的KV数据，减小数据规模，减少文件数量等。

levelDb的compaction机制和过程与Bigtable所讲述的是基本一致的，Bigtable中讲到三种类型的compaction: minor ， major和full。所谓minor Compaction，就是把memtable中的数据导出到SSTable文件中；major compaction就是合并不同层级的SSTable文件，而full compaction就是将所有SSTable进行合并。

LevelDb包含其中两种，minor和major。

朗格科技将为大家详细叙述其机理。

先来看看minor Compaction的过程。Minor compaction 的目的是当内存中的memtable大小到了一定值时，将内容保存到磁盘文件中，图8.1是其机理示意图。

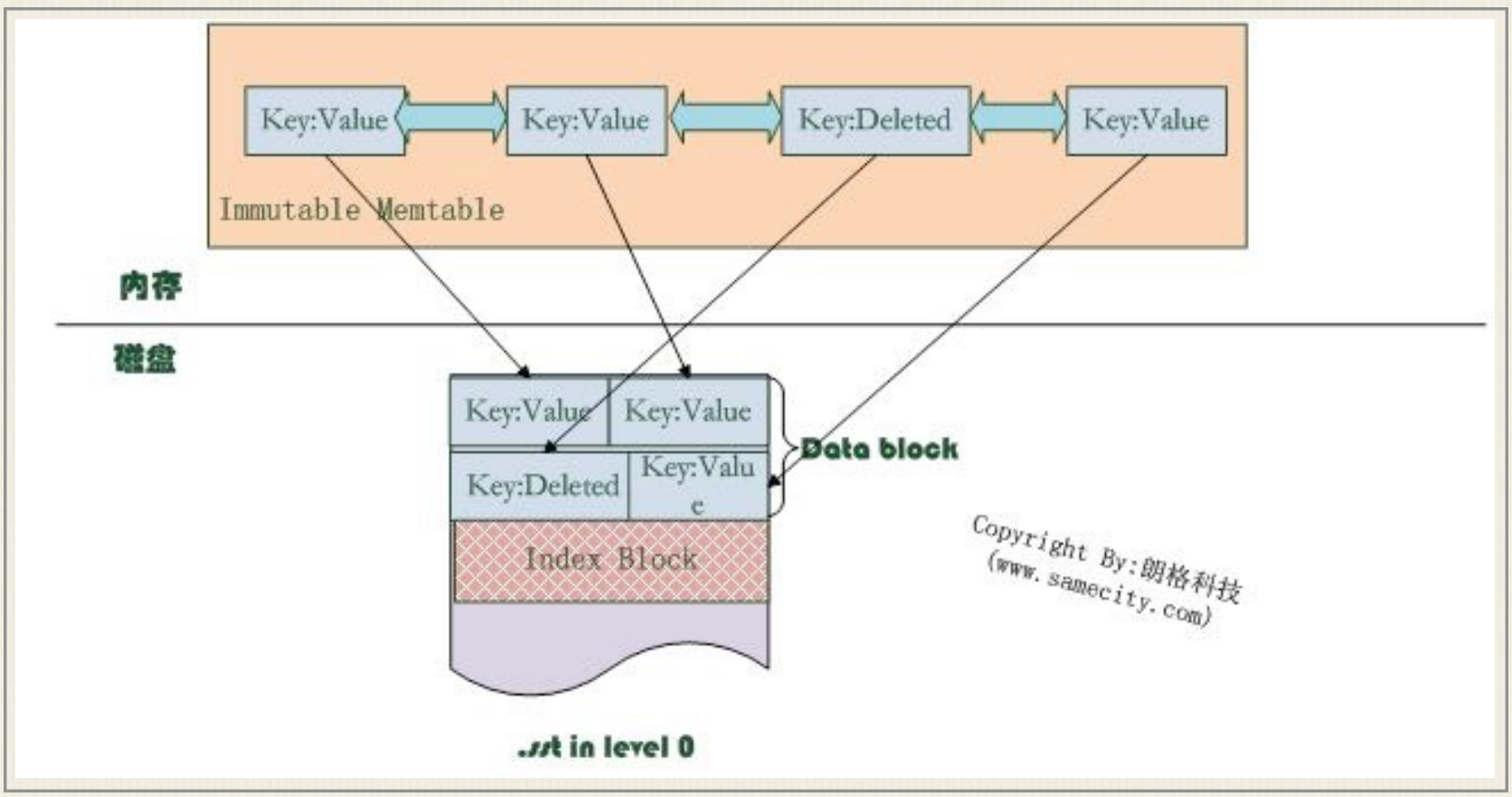


图8.1 minor compaction

从8.1可以看出，当memtable数量到了一定程度会转换为immutable memtable，此时不能往其中写入记录，只能从中读取KV内容。之前介绍过，immutable memtable其实是一个多层级队列SkipList，其中的记录是根据key有序排列的。所以这个minor compaction实现起来也很简单，就是按照immutable memtable中记录由小到大遍历，并依次写入一个level 0 的新建SSTable文件中，写完后建立文件的index 数据，这样就完成了一次minor compaction。从图中也可以看出，对于被删除的记录，在minor compaction过程中并不真正删除这个记录，原因也很简单，这里只知道要删掉key记录，但是这个KV数据在哪里?那需要复杂的查找，所以在 minor compaction的时候并不做删除，只是将这个key作为一个记录写入文件中，至于真正的删除操作，在以后更高层级的compaction中会去做。

当某个level下的SSTable文件数目超过一定设置值后，levelDb会从这个level的SSTable中选择一个文件（level>0），将其和高一层级的level+1的SSTable文件合并，这就是major compaction。

我们知道在大于0的层级中，每个SSTable文件内的Key都是由小到大有序存储的，而且不同文件之间的key范围（文件内最小key和最大key之间）不会有任何重叠。Level 0的SSTable文件有些特殊，尽管每个文件也是根据Key由小到大排列，但是因为level 0的文件是通过minor compaction直接生成的，所以任意两个level 0下的两个sstable文件可能再key范围上有重叠。所以在做major compaction的时候，对于大于level 0的层级，选择其中一个文件就行，但是对于level 0来说，指定某个文件后，本level中很可能有其他SSTable文件的key范围和这个文件有重叠，这种情况下，要找出所有有重叠的文件和level 1的文件进行合并，即level 0在进行文件选择的时候，可能会有多个文件参与major compaction。

levelDb在选定某个level进行compaction后，还要选择是具体哪个文件要进行compaction，levelDb在这里有个小技巧， 就是说轮流来，比如这次是文件A进行compaction，那么下次就是在key range上紧挨着文件A的文件B进行compaction，这样每个文件都会有机会轮流和高层的level 文件进行合并。

如果选好了level L的文件A和level L+1层的文件进行合并，那么问题又来了，应该选择level L+1哪些文件进行合并？levelDb选择L+1层中和文件A在key range上有重叠的所有文件来和文件A进行合并。

也就是说，选定了level L的文件A,之后在level L+1中找到了所有需要合并的文件B,C,D…..等等。剩下的问题就是具体是如何进行major 合并的?就是说给定了一系列文件，每个文件内部是key有序的，如何对这些文件进行合并，使得新生成的文件仍然Key有序，同时抛掉哪些不再有价值的KV数据。

图8.2说明了这一过程。

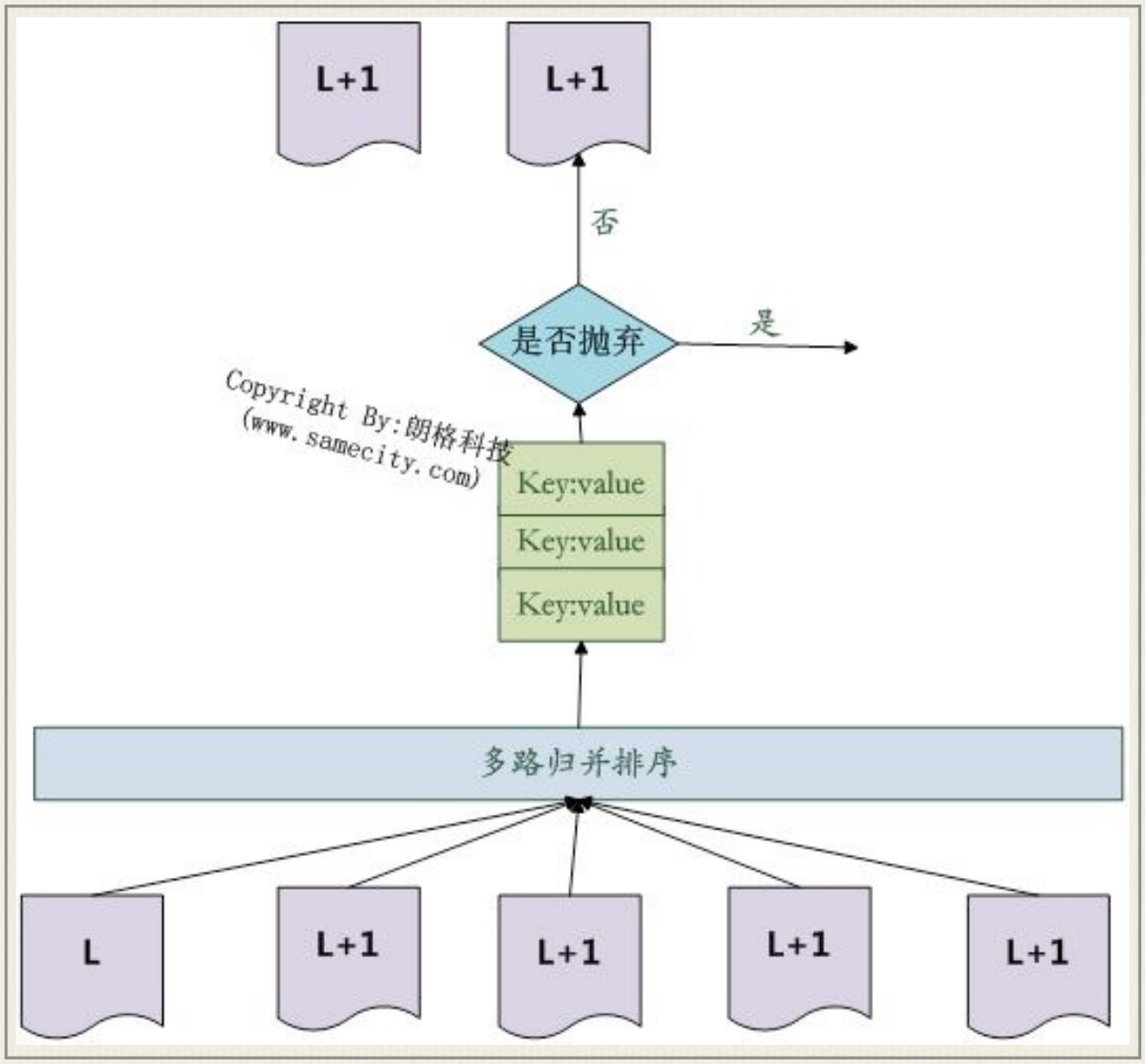


图8.2 SSTable Compaction

Major compaction的过程如下：对多个文件采用多路归并排序的方式，依次找出其中最小的Key记录，也就是对多个文件中的所有记录重新进行排序。之后采 取一定的标准判断这个Key是否还需要保存，如果判断没有保存价值，那么直接抛掉，如果觉得还需要继续保存，那么就将其写入level L+1层中新生成的一个SSTable文件中。就这样对KV数据一一处理，形成了一系列新的L+1层数据文件，之前的L层文件和L+1层参与 compaction 的文件数据此时已经没有意义了，所以全部删除。这样就完成了L层和L+1层文件记录的合并过程。

那么在major compaction过程中，判断一个KV记录是否抛弃的标准是什么呢？其中一个标准是:对于某个key来说，如果在小于L层中存在这个Key，那么这个 KV在major compaction过程中可以抛掉。因为我们前面分析过，对于层级低于L的文件中如果存在同一Key的记录，那么说明对于Key

来说，有更新鲜的 Value存在，那么过去的Value就等于没有意义了，所以可以删除。

是的，compacition的过程就是如此简单。

LevelDb日知录 之九: levelDb中的Cache

书接前文，前面讲过对于levelDb来说，读取操作如果没有在内存的memtable中找到记录，要多次进行磁盘访问操作。假设最优情况，即第一次就在level 0中最新的文件中找到了这个key，那么也需要读取2次磁盘，一次是将SSTable的文件中的index部分读入内存，这样根据这个index可以确定 key是在哪个block中存储；第二次是读入这个block的内容，然后在内存中查找key对应的value。

哎！

为什么?

怎么会这样?

这样效率有点低。

levelDb想到这点了。

它引入了Cache来部分解决读取数据效率低下的问题。

levelDb中引入了两个不同的Cache:Table Cache和Block Cache。

其中Block Cache是配置可选的，即在配置文件中指定是否打开这个功能。

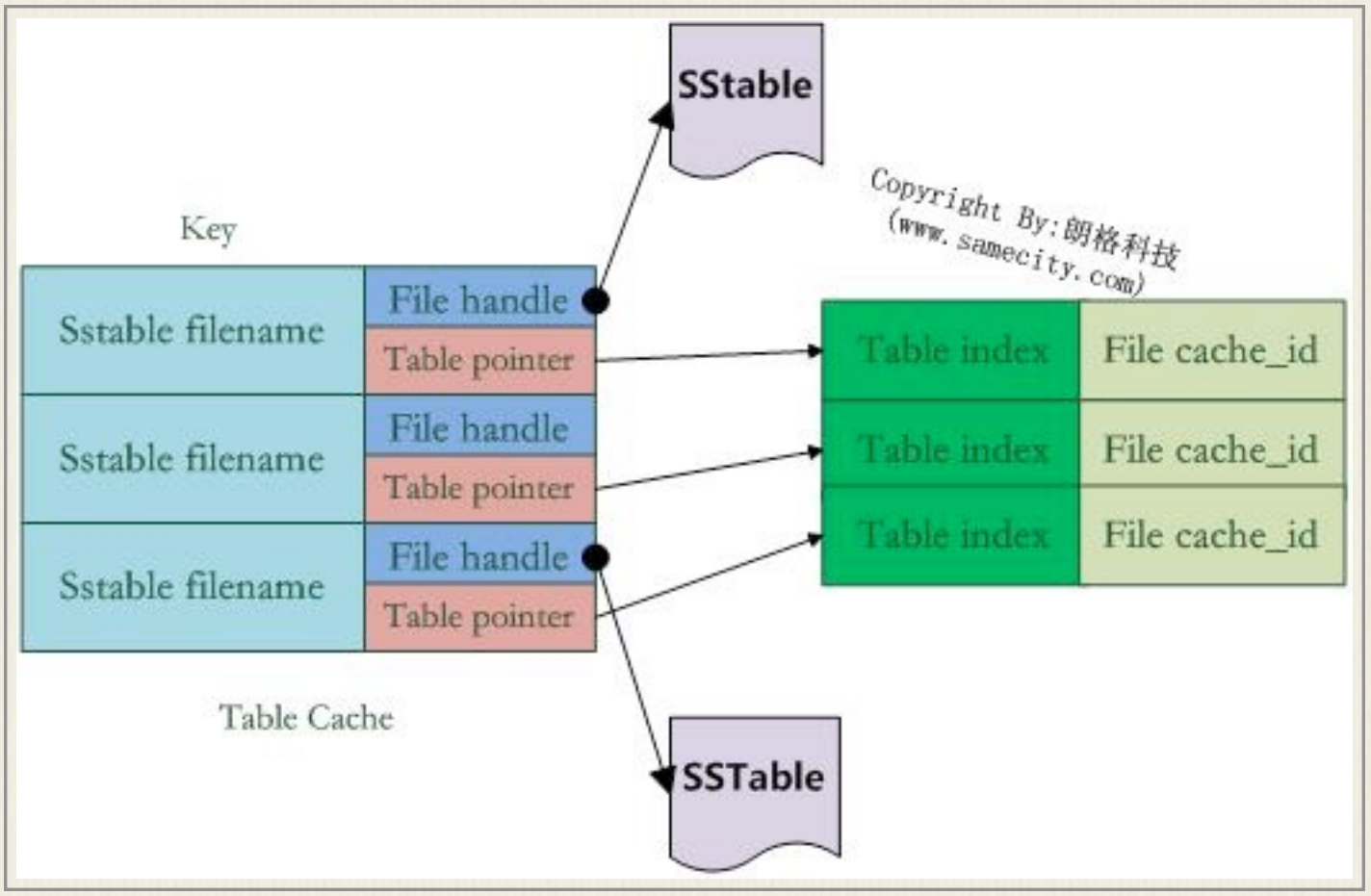


图9.1 table cache

图9.1是table cache的结构。在Cache中，key值是SSTable的文件名称，Value部分包含两部分，一个是指向磁盘打开的SSTable文件的文件指针，这是为了方便读取内容；另外一个是指向内存中这个SSTable文件对应的Table结构指针，table结构在内存中，保存了SSTable的 index内容以及用来指示block cache用的cache_id ,当然除此外还有其它一些内容。

比如在get(key)读取操作中，如果levelDb确定了key在某个level下某个文件A的key range范围内，那么需要判断是不是文件A真的包含这个KV。此时，levelDb会首先查找Table Cache，看这个文件是否在缓存里，如果找到了，那么根据index部分就可以查找是哪个block包含这个key。如果没有在缓存中找到文件，那么打 开SSTable文件，将其index部分读入内存，然后插入Cache里面，去index里面定位哪个block包含这个Key 。

如果确定了文件哪个block包含这个key ，那么需要读入block内容，这是第二次读取。

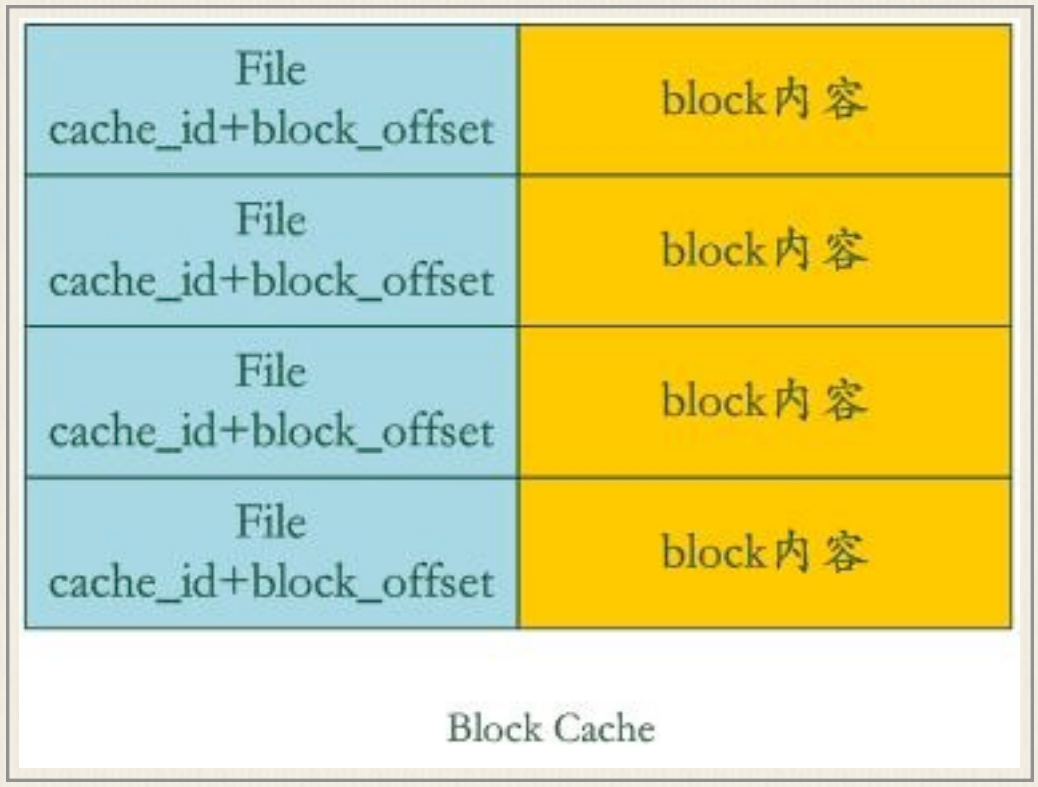


图9.2 block cache

Block Cache是为了加快这个过程的，图9.2是其结构示意图。其中的key是文件的cache_id加上这个block在文件中的起始位置block_offset。而value则是这个Block的内容。

如果levelDb发现这个block在block cache中，那么可以避免读取数据，直接在cache里的block内容里面查找key的value就行，如果没找到呢？那么读入block内容并把它插入block cache中。

levelDb就是这样通过两个cache来加快读取速度的。从这里可以看出，如果读取的数据局部性比较好，也就是说要读的数据大部分在cache里面都 能读到，那么读取效率应该还是很高的，而如果是对key进行顺序读取效率也应该不错，因为一次读入后可以多次被复用。但是如果是随机读取，您可以推断下其 效率如何。

LevelDb日知录之十 Version、VersionEdit、VersionSet

Version 保存了当前磁盘以及内存中所有的文件信息，一般只有一个Version叫做"current" version（当前版本）。Leveldb还保存了一系列的历史版本，这些历史版本有什么作用呢？

当一个Iterator创建后，Iterator就引用到了current version(当前版本)，只要这个Iterator不被delete那么被Iterator引用的版本就会一直存活。这就意味着当你用完一个Iterator后，需要及时删除它。

当一次Compaction结束后（会生成新的文件，合并前的文件需要删除），Leveldb会创建一个新的版本作为当前版本，原先的当前版本就会变为历史版本。

VersionSet 是所有Version的集合，管理着所有存活的Version。

VersionEdit 表示Version之间的变化，相当于delta 增量，表示有增加了多少文件，删除了文件。下图表示他们之间的关系。

Version0 +VersionEdit-->Version1

VersionEdit会保存到MANIFEST文件中，当做数据恢复时就会从MANIFEST文件中读出来重建数据。

leveldb的这种版本的控制，让我想到了双buffer切换，双buffer切换来自于图形学中，用于解决屏幕绘制时的闪屏问题，在服务器编程中也有用处。

比如我们的服务器上有一个字典库，每天我们需要更新这个字典库，我们可以新开一个buffer，将新的字典库加载到这个新buffer中，等到加载完毕，将字典的指针指向新的字典库。

leveldb的version管理和双buffer切换类似，但是如果原version被某个iterator引用，那么这个version会一直保持，直到没有被任何一个iterator引用，此时就可以删除这个version。

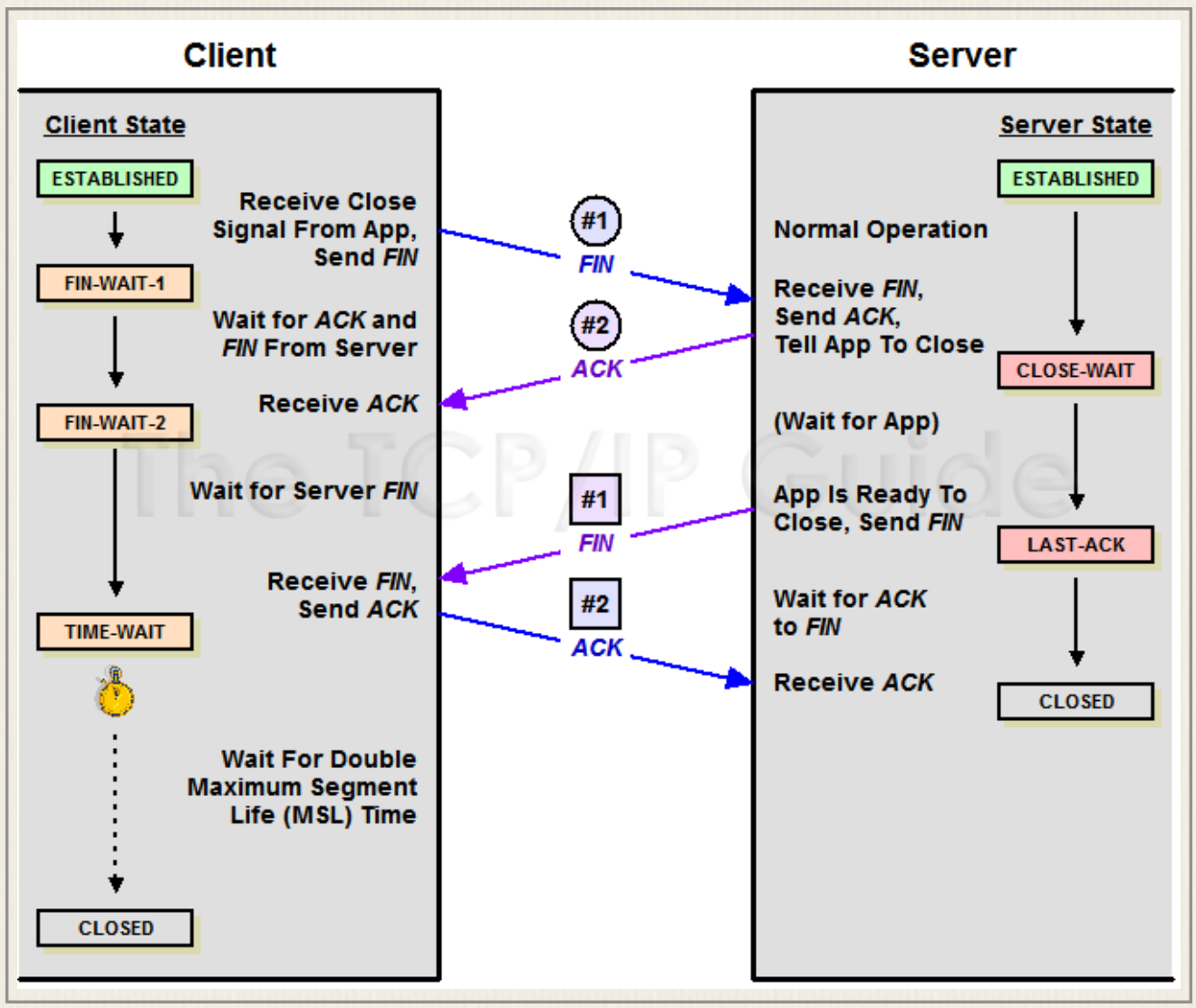
原文链接：http://www.samecity.com/blog/Index.asp?SortID=12

关于FIN_WAIT1

作者：火丁笔记

前些天，一堆人在 TCPCopy 社区里闲扯蛋，有人提了一个问题：FIN_WAIT1 能持续多久？引发了一场讨论，期间我得到斌哥和多位朋友的点化，受益良多。

让我们热热身，通过一张旧图来回忆一下 TCP 关闭连接时的情况：



TCP Close

看图可知，主动关闭的一方发出 FIN，同时进入 FIN_WAIT1 状态，被动关闭的一方响应 ACK，从而使主动关闭的一方迁移至 FIN_WAIT2 状态，接着被动关闭的一方同样会发出 FIN，主动关闭的一方响应 ACK，同时迁移至 TIME_WAIT 状态。

回到开头的问题：FIN_WAIT1 能持续多久？一般情况下，服务器间的 ACK 确认是非常快的，以至于我们凭肉眼往往观察不到 FIN_WAIT1 的存在，不过网上也有很多案例表明在某些情况下 FIN_WAIT1 会持续很长时间，从而诱发问题。

最常见的误解是认为 tcp_fin_timeout 控制 FIN_WAIT1 的过期，从名字上看也很像，但实际上它控制的是 FIN_WAIT2 的过期时间，官方文档是这样说的：

The length of time an orphaned (no longer referenced by any application) connection will remain in the FIN_WAIT_2 state before it is aborted at the local end. While a perfectly valid “receive only” state for an un-orphaned connection, an orphaned connection in FIN_WAIT_2 state could otherwise wait forever for the remote to close its end of the connection.
Cf. tcp_max_orphans
Default: 60 seconds

让我们通过一个实验来说明问题（服务端：10.16.15.107；客户端：10.16.15.109）：

1. 在服务端监听 1234 端口：「nc -l 1234」
2. 在客户端连接服务端：「nc 10.16.15.107 1234」
此时客户端连接进入 ESTABLISHED 状态
3. 在服务端拦截响应：「iptables -A OUTPUT -d 10.16.15.109 -j DROP」
4. 在客户端开启抓包：「tcpdump -nn -i any port 1234」
5. 在客户端通过「ctrl + c」断开连接
此时客户端连接进入 FIN_WAIT1 状态

随时可以通过「netstat -ant | grep :1234」来观察状态，最终抓包结果如下：

```
15:26:51.640810 IP 10.16.15.109.59837 > 10.16.15.107.1234: F 3102924593:3102924593(0)
15:26:51.843484 IP 10.16.15.109.59837 > 10.16.15.107.1234: F 0:0(0) ack 1 win 46 <nop
15:26:52.251744 IP 10.16.15.109.59837 > 10.16.15.107.1234: F 0:0(0) ack 1 win 46 <nop
15:26:53.037756 IP 10.16.15.109.59837 > 10.16.15.107.1234: F 0:0(0) ack 1 win 46 <nop
15:26:54.670616 IP 10.16.15.109.59837 > 10.16.15.107.1234: F 0:0(0) ack 1 win 46 <nop
15:26:57.936319 IP 10.16.15.109.59837 > 10.16.15.107.1234: F 0:0(0) ack 1 win 46 <nop
15:27:04.467720 IP 10.16.15.109.59837 > 10.16.15.107.1234: F 0:0(0) ack 1 win 46 <nop
15:27:17.530527 IP 10.16.15.109.59837 > 10.16.15.107.1234: F 0:0(0) ack 1 win 46 <nop
15:27:43.656111 IP 10.16.15.109.59837 > 10.16.15.107.1234: F 0:0(0) ack 1 win 46 <nop
```

TCP Fin

第一个 FIN 是我们按「ctrl + c」断开连接时触发的，因为我们在服务端通过 iptables 拦截了发送给客户端的响应，所以对应的 ACK 被丢弃，随后执行了若干次重试。

此外，通过观察时间我们还能发现，第一次重试在 200ms 左右；第二次是在 400ms 左右；第三次是在 800ms 左右；以此类推，每次的时间翻倍。

实际上，控制这一行为的关键参数是 tcp_orphan_retries，官方文档是这样说的：

This value influences the timeout of a locally closed TCP connection, when RTO retransmissions remain unacknowledged. See tcp_retries2 for more details.
The default value is 8. If your machine is a loaded WEB server, you should think about lowering this value, such sockets may consume significant resources. Cf. tcp_max_orphans.

如果你用 sysctl 查询 tcp_orphan_retries 是 0，那么实际等同于 8，看代码：

/ Calculate maximal number or retries on an orphaned socket. */*

```
static int tcp_orphan_retries(struct sock *sk, int alive)
{
    int retries = sysctl_tcp_orphan_retries; /* May be zero. */

    /* We know from an ICMP that something is wrong. */
    if (sk->sk_err_soft && !alive)
        retries = 0;

    /* However, if socket sent something recently, select some safe
     * number of retries. 8 corresponds to >100 seconds with
     * minimal RTO of 200msec. */
    if (retries == 0 && alive)
        retries = 8;
    return retries;
}
```

于是乎我们可以得出结论，如果你的服务器负载较重，有很多 FIN_WAIT1，那么可以考虑通过降低 tcp_orphan_retries 来解决问题，具体设置多少视网络条件而定。

问题分析到这里原本可以完美谢幕，但是因为 TCP 有缺陷，导致 FIN_WAIT1 可能被用来发起 DoS 攻击，所以我们就再唠十块钱儿的，看看到底是怎么回事儿：

假设服务端上有一个大文件，攻击者连接服务端发起请求，但是却不接收数据，于是乎就造成一种现象：客户端接收队列满，导致服务端不得不通过 「zero window probes」来循环检测客户端是否有可用空间，以至

于 tcp_orphan_retries 也没有用，因为服务端活活被憋死了，发不出 FIN 来，从而永远卡在 FIN_WAIT1。演示代码如下：

```
#!/usr/bin/env python

import socket
import time

host = 'www.domain.com'
port = 80
path = '/a/big/file'

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((host, port))
sock.send("GET %s HTTP/1.0\r\nHost: %s\r\n\r\n" % (path, host))

time.sleep(1000)
```

说明：通常文件大小以 100K 为佳，具体取决于 tcp_rmem / tcp_wmem 的大小。

怎么办？病急乱投医，重启服务！可惜没用，因为 FIN_WAIT1 已经脱离的服务的管辖范围，所以重启服务是没有用的，如果一定要重启，你只能重启服务器！

好在内核已经考虑到了此类问题，它提供了 tcp_max_orphans 参数，用来控制 orphans 的最大值，需要注意的是，和用来控制 TIME_WAIT 的最

大值的 tcp_max_tw_buckets 参数一样，除非你遇到了 DoS 攻击，否则最好不要降低它。

花絮：我曾经试图寻找一些工具来杀掉 FIN_WAIT1 连接，如果你要杀掉一个 TCP 连接，那么需要知道相应的 ACK 和 SEQ，然后才可以 RESET 连接。为了获取 ACK 和 SEQ，一些工具采用的是被动机制，它通过监听匹配的数据包来获取需要的数据，代表是 tcpkill；另一些工具采用的是主动机制，它通过伪造请求来获取需要的数据，代表是 killcx，如果有兴趣的话不妨试试它们。

最后，再次感谢 TCPCopy 社区！如果你从本文学到些许知识，那么这份荣幸属于 TCPCopy社区，如果你在本文发现谬误之处，那么全因本人笨拙，还望不吝赐教。

原文链接：http://huoding.com/2014/11/06/383

回收站功能在 Linux 中的实现

作者：李朝阳

本文仿照 Windows 回收站的功能，运用 Bash 脚本在 Linux 上做了实现，创建 delete 脚本代替 rm 命令对文件或目录进行删除操做。该脚本实现了以下功能：对大于 2G 的文件或目录直接删除，否则放入$HOME/trash 目录下；恢复 trash 目录中的被删除文件到原目录下；文件存放在 trash 目录中超过七天被自动删除。

概述

删除是危险系数很高的操作，一旦误删可能会造成难以估计的损失。在 Linux 系统中这种危险尤为明显，一条简单的语句：rm –rf /* 就会把整个系统全部删除，而 Linux 并不会因为这条语句的不合理而拒绝执行。 在 Windows 中，为了防止误删，系统提供了回收站功能。用户在执行删除操作后，文件并不会直接从硬盘中删除，而是被放到回收站中。在清空回收站前，如果发现有文件被误 删，用户可以将回收站中的文件恢复到原来的位置。而 Linux 并没有提供类似功能，删除命令 rm 一旦确认执行，文件就会直接从系统中删除，很难恢复。

回收站构成

本文共用三个脚本实现了回收站的主要功能：Delete 脚本、logTrashDir 脚本和 restoreTrash 脚本。其中 Delete 脚本是核心脚本，其作用是重新封装 rm 命令。相对于 rm 的直接删除，该命令会先将文件或目录移动到 $home/trash 目录下。如果用户想要将文件直接删除，可以用 -f 选项，delete 脚本会直接调用 rm –f 命令将文件从硬盘上删除。logTrashDir 脚本用

于将被删除文件的信息记录到 trash 目录下的一个隐藏文件中。restoreTrash 脚本用来将放入 trash 中的文件或目录重新恢复到原路径下。在 Linux 系统中，只要将这三个脚本放到/bin/目录下，并用 chmod +X filename 赋予可执行权限，即可直接使用。下面将介绍每个脚本的主要部分

Delete 脚本

创建目录

首先要创建目录来存放被删除的文件，本文在用户根目录$HOME 下建立 trash 目录来存放文件。具体代码如下：

清单 1.创建回收站目录

C代码

```
1.  realrm="/bin/rm"
2.  if [ ! -d ~/trash ]
3.   then
4.       mkdir -v ~/trash
5.       chmod 777 ~/trash
6.  fi
```

如上所示，先判断目录是否已建立，如未建立，即第一次运行该脚本，则创建 trash 目录。变量 realrm 存放了 Linux 的 rm 脚本位置，用于在特定条件下调用以直接删除文件或目录。

输出帮助信息

该脚本在用户仅输入脚本名而未输入参数执行时，输出简要帮助信息，代码如下：

清单 2.输出帮助信息

C代码

```
1.    if [ $# -eq 0 ]
2.      then
3.          echo "Usage:delete file1 [file2 file3....]"
4.
echo "If the options contain -f,then the script will exec 'rm' directly"
```

如代码所示，该脚本的运用格式是 delete 后跟要删除的文件或目录的路径，中间用空格隔开。

直接删除文件

有些用户确认失效并想直接删除的文件，不应放入回收站中，而应直接从硬盘中删除。Delete 脚本提供了-f 选项来执行这项操作：

清单 3.直接删除文件

C代码

```
1.    while getopts "dfiPRrvW" opt
```

```
2.          do
3.            case $opt in
4.                f)
5.                  exec $realrm "$@"
6.                  ;;
7.                *)
8.
9.                  # do nothing
10.                 ;;
11.           esac
12.         done
```

如果用户在命令中加入了-f 选项，则 delete 脚本会直接调用 rm 命令将文件或目录直接删除。如代码中所示，所有的参数包括选项都会传递给 rm 命令。所以只要选项中包括选项-f 就等于调用 rm 命令，可以使用 rm 的所有功能。如：delete –rfv filename 等于 rm –rfv filename。

用户交互

需要与用户确认是否将文件放入回收站。相当于 Windows 的弹窗提示，防止用户误操作。

清单 4.用户交互

C代码

```
1.
echo -ne "Are you sure you want to move the files to the trash?[Y/N]:\a"
2.     read reply
3.    if [ $reply = "y" -o $reply = "Y" ]
4.      then #####
```

判断文件类型并直接删除大于 2G 文件

本脚本只对普通文件和目录做操作，其他类型文件不做处理。先对每个参数做循环，判断他们的类型，对于符合的类型再判断他们的大小是否超过2G，如果是则直接从系统中删除，避免回收站占用太大的硬盘空间。

清单 5.删除大于 2G 的文件

C代码

```
1.    for file in $@
2.     do
3.    if [ -f "$file" –o –d "$file" ]
4.    then
5.    if [ -f "$file" ] && [ `ls –l $file|awk '{print $5}'` -gt 2147483648 ]
6.       then
7.           echo "$file size is larger than 2G,will be deleted directly"
8.              `rm –rf $file`
9.    elif [ -d "$file" ] && [ `du –sb $file|awk '{print $1}'` -
gt 2147483648 ]
```

```
10.     then
11.
echo "The directory:$file is larger than 2G,will be deleted directly"
12.         `rm –rf $file`
```

如以上代码所示，该脚本用不同的命令分别判断目录和文件的大小。鉴于目录的大小应该是包含其中的文件以及子目录的总大小，所以运用了'du -sb'命令。两种情况都使用了 awk 来获取特定输出字段的值来作比较。

移动文件到回收站并做记录

该部分是 Delete 脚本的主要部分，主要完成以下几个功能

- 获取参数的文件名。因为用户指定的参数中可能包含路径，所以要从中获取到文件名，用来生成 mv 操作的参数。该脚本中运用了字符串正则表达式'${file##*/}'来获取。
- 生成新文件名。在原文件名中加上日期时间后缀以生成新的文件名，这样用户在浏览回收站时，对于每个文件的删除日期即可一目了然。
- 生成被删文件的绝对路径。为了以后可能对被删文件进行的恢复操作，要从相对路径生成绝对路径并记录。用户输入的参数可能有三种情况：只包含文件名的相对路径，包含点号的相对路径以及绝对路径，脚本中用字符串处理对三种情况进行判断，并进行相应的处理。
- 调用 logTrashDir 脚本，将回收站中的新文件名、原文件名、删除时间、原文件绝对路径记录到隐藏文件中
- 将文件通过 mv 命令移动到 Trash 目录下。

详细代码如下所示：

清单 6.移动文件到回收站并做记录

C代码

```
1.  now=`date +%Y%m%d_%H_%M_%S`
2.  filename="${file##*/}"
3.  newfilename="${file##*/}_${now}"
4.  mark1="."
5.  mark2="/"
6.  if  [ "$file" = ${file/$mark2} ]
7.   then
8.    fullpath="$(pwd)/$file"
9.  elif [ "$file" != ${file/$mark1} ]
10.  then
11.   fullpath="$(pwd)${file/$mark1}"
12. else
13.   fullpath="$file"
14. fi
15. echo "the full path of this file is :$fullpath"
16. if mv -f $file ~/trash/$newfilename
17.  then
18.   $(/logTrashDir "$newfilename $filename $now $fullpath")
19.    echo "files: $file is deleted"
20.  else
21.   echo "the operation is failed"
22.  fi
```

logTrashDir 脚本

该脚本较简单，仅是一个简单的文件写入操作，之所以单独作为一个脚本，是为了以后扩展的方便，具体代码如下：

清单 7.logTrashDir 代码

C代码

```
1.  if [ ! -f ~/trash/.log ]
2.    then
3.      touch ~/trash/.log
4.      chmod 700~/trash/.log
5.  fi
6.    echo $1 $2 $3 $4>> ~/trash/.log
```

该脚本先建立.log 隐藏文件，然后往里添加删除文件的记录。

restoreTrash 脚本

该脚本主要完成以下功能：

- 从.log 文件中找到用户想要恢复的文件对应的记录。此处依然使用 awk，通过正表达式匹配找到包含被删除文件名的一行
- 从记录中找到记录原文件名的字段，以给用户提示
- 将回收站中的文件移动到原来的位置，在这里运用了 mv –b 移动文件，之所以加入-b 选项是为了防止原位置有同名文件的情况。
- 将.log 文件中与被恢复文件相对应的记录删除

清单 8.获取相应记录

C代码

```
1.  originalPath=$(awk /$filename/'{print $4}' "$HOME/trash/.log")
```

清单 9.查找原文件名及现文件名字段

C代码

```
1.  filenameNow=$(awk /$filename/'{print $1}' ~/trash/.log)
2.  filenamebefore=$(awk /$filename/'{print $2}' ~/trash/.log)
3.
echo "you are about to restore $filenameNow,original name is $filenamebe
fore"
4.  echo "original path is $originalPath"
```

清单 10.恢复文件到原来位置并删除相应记录

C代码

```
1.  echo "Are you sure to do that?[Y/N]"
2.  read reply
3.  if [ $reply = "y" ] || [ $reply = "Y" ]
4.   then
5.  $(mv -b "$HOME/trash/$filename" "$originalPath")
6.  $(sed -i /$filename/'d' "$HOME/trash/.log")
7.  else
8.    echo "no files restored"
9.  fi
```

自动定期清理 trash 目录

因为 delete 操作并不是真正删除文件，而是移动操作，经过一段时间的积累，trash 目录可能会占用大量的硬盘空间，造成资源浪费，所以定期自

动清理 trash 目录下的文件是必须得。本文的清理规则是：在回收站中存在 7 天以上的文件及目录将会被自动从硬盘中删除。运用的工具是 Linux 自带的 crontab。

Crontab 是 Linux 用来定期执行程序的命令。当安装完成操作系统之后，默认便会启动此任务调度命令。Crontab 命令会定期检查是否有要执行的工作，如果有要执行的工作便会自动执行该工作。而 Linux 任务调度的工作主要分为以下两类：

1、系统执行的工作：系统周期性所要执行的工作，如备份系统数据、清理缓存

2、个人执行的工作：某个用户定期要做的工作，例如每隔 10 分钟检查邮件服务器是否有新信，这些工作可由每个用户自行设置。

首先编写 crontab 执行时要调用的脚本 cleanTrashCan.如清单 10 所示，该脚本主要完成两项功能：

- 判断回收站中的文件存放时间是否已超过 7 天，如果超过则从回收站中删除。
- 将删除文件在.log 文件中相应的记录删除，保持其中数据的有效性，提高查找效率。

清单 11.删除存在回收站超过 7 天的文件并删除.log 中相应记录

C代码

```
1.  arrayA=($(find ~/trash/* -mtime +7 | awk '{print $1}'))
2.      for file in ${arrayA[@]}
3.       do
4.         $(rm -rf "${file}")
5.         filename="${file##*/}"
6.         echo $filename
7.         $(sed -i /$filename/'d' "$HOME/trash/.log")
```

```
8.      done
```

脚本编写完成后通过 chmod 命令赋予其执行权限，然后运过 crontab –e 命令添加一条新的任务调度：

C代码

```
1.  10 18 * * * /bin/ cleanTrashCan
```

该语句的含义为，在每天的下午 6 点 10 分执行 cleanTrashCan 脚本

通过这条任务调度，trash 的大小会得到有效的控制，不会持续增大以致影响用户的正常操作。

实际应用

首先要将 delete 脚本，logTrashDir 脚本，restoreTrash 脚本和 clean-TrashCan 放到/bin 目录下，然后用 chmod +x delete restoreTrash log-TrashDir cleanTrashCan 命令赋予这三个脚本可执行权限。

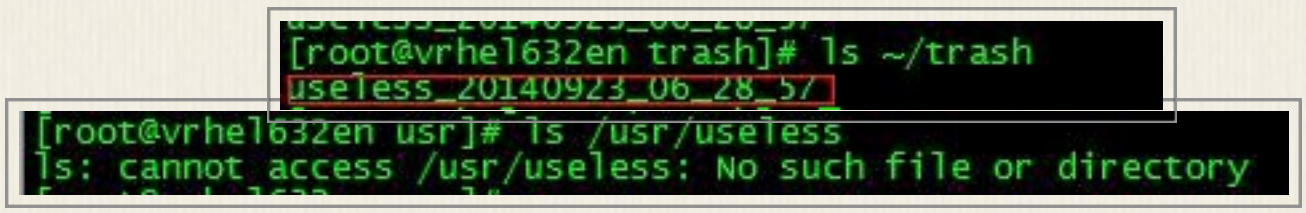
运用 delete 脚本删除文件，例如要删除在/usr 目录下的 useless 文件。根据用户目前所在的位置，可以用相对路径或绝对路径来指定参数，如：delete useless，delete ./useless 或者 delete /usr/useless。执行过程如图 1 所示：

图 1.delete 脚本执行过程

```
[root@vrhel632en usr]# touch useless
[root@vrhel632en usr]# delete useless
Are you sure you want to move the files to the trash?[Y/N]:useless
y
the full path of this file is :/usr/useless
files: useless is deleted
```

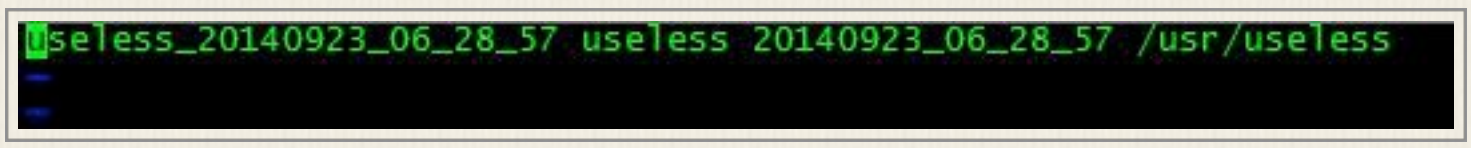
执行之后，useless 文件会从原目录中删除，被移动到$HOME/trash 下，并被重命名，如图 2.所示：

图 2.回收站目录



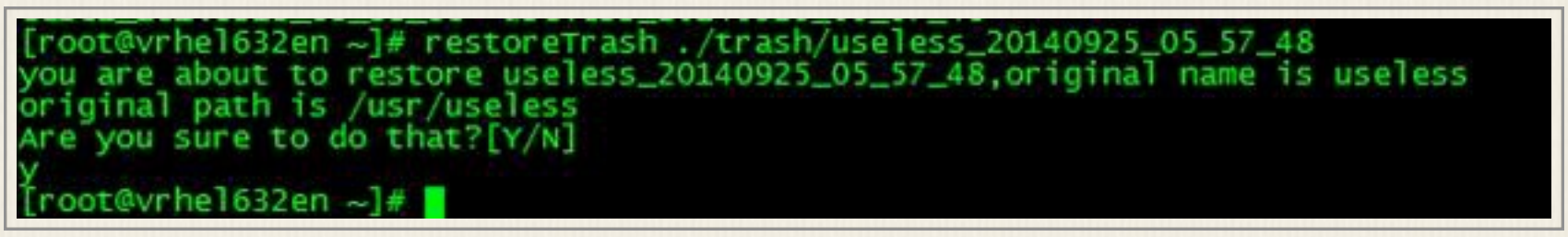
生成的.log 记录如图 3.所示：

图 3.log 记录



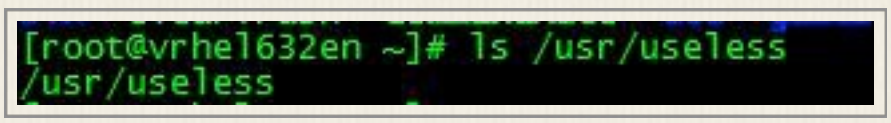
如果用户在七天之内发现该文件还有使用价值，则可以使用 restoreTrash 命令将被删除文件恢复到原路径下：restoreTrash ~/trash/useless_20140923_06_28_57。具体执行情况如图 4 所示：

图 4.restoreTrash 脚本执行情况



查看/usr 目录，可以发现 useless 文件已经被恢复至此。

图 5.useless 文件被恢复



总结

本文仿照 Windows 中回收站的功能，在 Linux 中做了实现，可以有效的防止由于误删而造成的损失。读者只需要将四个脚本拷到/bin 目录下，并配置 crontab 即可使用 Linux 版回收站。

原文链接：http://www.ibm.com/developerworks/cn/linux/1410_licy_linuxtrash/

iOS APP 架构漫谈二

作者：不会开机的男孩

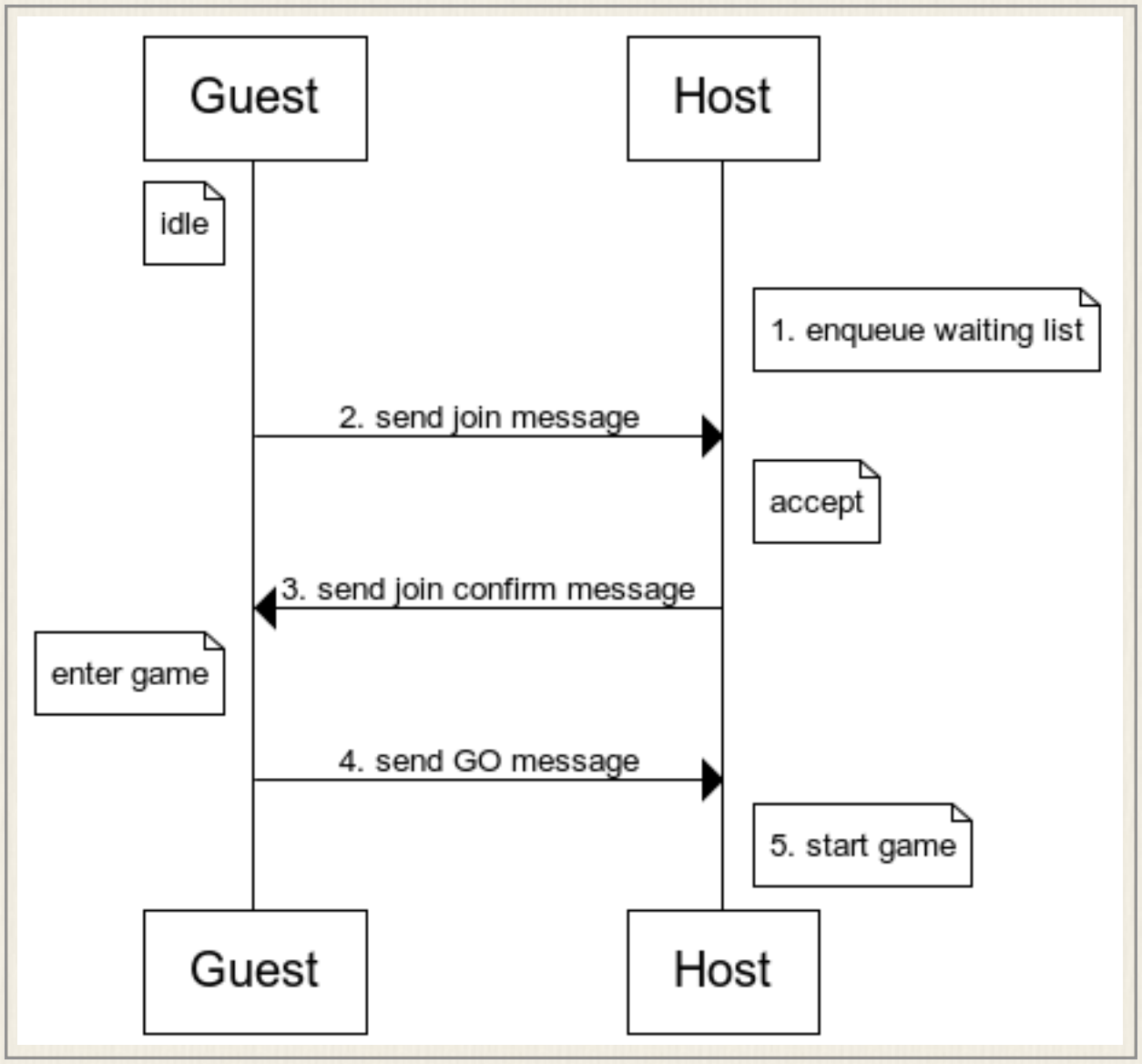
背景

首先看下我们的使用场景，假如我们需要设计一套联网对战的小游戏。第一个难题可能是如何建立一个通道，让2个手机相互发送消息。这里我并不打算引入server端开发，希望只是通过客户端来实现这个逻辑，这里使用LeanCloud API来简化这个过程。这样我们可以暂时不考虑技术细节，直接站在业务角度去思考如何建立这个游戏。

业务场景–邀请

正式开始游戏之前，总会有一个邀请的环节。假如我们有2个用户，分别是Host，Guest。Host创建游戏，Guest加入游戏。游戏的整个流程和我们平时玩的对战游戏流程并没有多大不同。

1. Host创建游戏，他就相当于进入一个等待队列里面。
2. Guest加入游戏，他从等待队列中找到一个匹配，比如Host。然后对Host发送join message
3. Host会收到很多join message。由于我们只是选择1vs1。这里假定Host同意Guest加入游戏。Host向Guest发送join confirm message
4. Guest收到join confirm message, 向Host发送Go消息，表示Guest已经进入游戏
5. Host收到Go消息。也进入游戏。

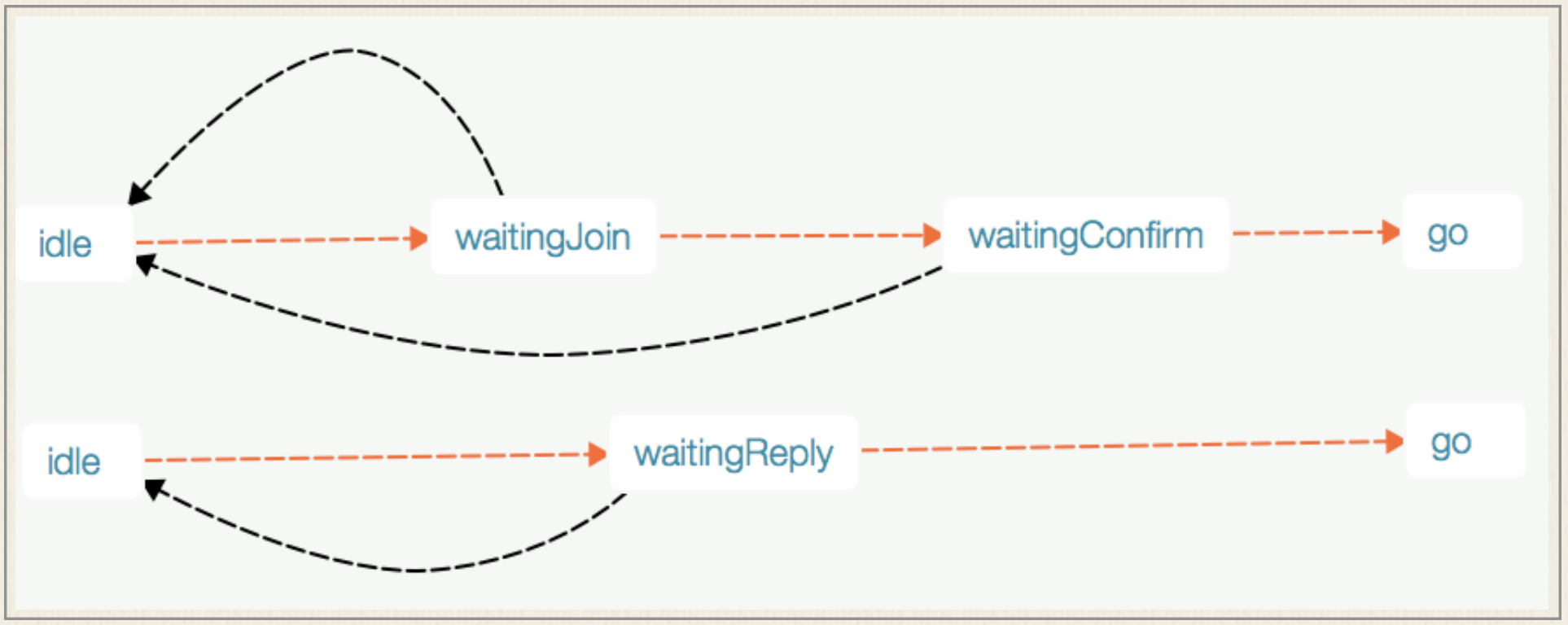


具体实现业务逻辑

现在的构想的逻辑只有5步，但其实还会包含很多逻辑，比如超时机制，重发机制。由于中间状态很多，还可能有我们没有想到过的问题。在面对这种复杂逻辑时，会通过状态机来帮助我们理顺逻辑。这时，我们脑中思考的业务其实是一个状态到一个状态的图。 如下

上半部分是游戏的创建者，下半部分是游戏的加入者。

一开始，尽量简化模型，这里红色剪头表示我们的正确主流路线，黑色表现出错路线。也就是说，一旦错误，就回到原始Idle状态。



开始写代码

在想清楚所有逻辑，并考虑清楚正常路线和错误路线之后，就可以开始写代码了。为了方便，这里直接使用第三方的状态机框架TransitionKit。

定义State（HOST）

```
TKState *idleState = [TKState stateWithName:@"idle"];
TKState *waitingJoinState = [TKState stateWithName:@"waitingJoin"];
TKState *waitingConfirmState = [TKState stateWithName:@"waitingConfirm"];
TKState *goState = [TKState stateWithName:@"go"];

[waitingConfirmState setDidEnterStateBlock:^(TKState *state, TKTransition *transition) {
    [selfWeak sendJoinConfirm];
}];
```

```
    [goState setDidEnterStateBlock:^(TKState *state, TKTransition *transi-
tion) {
        NSLog(@"happy ending");

        [SVProgressHUD showSuccessWithStatus:@"ok"];
    }];
```

定义Event（HOST）

Event 是建立State到State的路径

```
    TKEvent *waitingJoinEvent = [TKEvent
eventWithName:CUHostGameManagerWaitingJoinEvent
                                transitioningFromStates:@[idleState]
                                toState:waitingJoinState];

    TKEvent *receiveInviteEvent = [TKEvent
eventWithName:CUHostGameManagerReceiveInviteEvent
                                transitioningFromStates:@[waitingJoinState]
                                toState:waitingConfirmState];

    TKEvent *receiveConfirmEvent = [TKEvent
eventWithName:CUHostGameManagerReceiveConfirmEvent
                                transitioningFromStates:@[waitingConfirmState]
                                toState:goState];
```

```
TKEvent *disconnectedEvent = [TKEvent
eventWithName:CUHostGameManagerDisconnectedEvent
                                transitioningFromStates:nil
                                              toState:idleState];
```

定义过程（**HOST**）

```
- (void)startGame {

    NSAssert(self.session.peerId != nil, @"");

    //这里，如果不是idle，我们切换状态机到idle
    if (![self.stateMachine.currentState.name isEqual:@"idle"]) {
        [self fireEvent:CUHostGameManagerDisconnectedEvent
userInfo:nil];
    }

    //这里调用LeanCloud 入队
    AVObject *waitingId = [AVObject
objectWithClassName:@"waiting_join_Ids"];
    [waitingId setObject:self.session.peerId forKey:@"peerId"];
    [waitingId saveInBackgroundWithBlock:^(BOOL succeeded, NSEr-
ror *error) {
        //enqueue 之后，进入waitingJoin状态
```

```
            [self fireEvent:CUHostGameManagerWaitingJoinEvent
userInfo:nil];
        }];
    }

    - (void)sendJoinConfirm {
        //发送加入确认消息给Guest
        AVMessage *message = [AVMessage
messageForPeerWithSession:self.session
                                                toPeerId:self.peerId
                                                 payload:@"join_confirm"];
        [self.session sendMessage:message transient:YES];
    }

    - (void)session:(AVSession *)session
didReceiveMessage:(AVMessage *)message
    {
        if ([message.payload isEqualToString:@"join"]) {
            //收到Join（邀请）之后，发送确认消息
            self.peerId = message.fromPeerId;

            //因为LeanCloud的API比较挫，watch 之后才能发送消息，但是我们不知道什么时候才watch成功。。。。
            //好在只是demo，我们只好用这种方式work around，延迟2s发送消息
```

```
            [NSObject cancelPreviousPerformRequestsWithTarget:self
selector:@selector(sendInviteConfirmRequest:) object:nil];
            [self performSelector:@selector(sendInviteConfirmRequest:)
                    withObject:@[message.fromPeerId]
                    afterDelay:2.0f];
        }
        else if ([message.payload isEqualToString:@"go"]) {
            //收到go消息，流程结束
            [self fireEvent:CUHostGameManagerReceiveConfirmEvent
userInfo:nil];
        }
    }
    - (void)sendInviteConfirmRequest:(NSArray *)watchPeerIds {
        [self.session watchPeerIds:watchPeerIds];
        [self fireEvent:CUHostGameManagerReceiveInviteEvent
userInfo:nil];
    }
```

定义**State**（**Guest**）

```
    TKState *idleState = [TKState stateWithName:@"idle"];
    TKState *waitingReplyState = [TKState
stateWithName:@"waitingReply"];
    TKState *goState = [TKState stateWithName:@"go"];
```

```
    [waitingReplyState setWillEnterStateBlock:^(TKState *state, TKTransition *transition) {
        [selfWeak searchingGames];
    }];

    [goState setDidEnterStateBlock:^(TKState *state, TKTransition *transition) {
        [selfWeak sendGo];
        NSLog(@"happy ending");
        [SVProgressHUD showSuccessWithStatus:@"ok"];
    }];
```

定义Event（Guest）

```
    TKEvent *searchingEvent = [TKEvent
eventWithName:CUGestGameManagerSearchingEvent
                        transitioningFromStates:@[idleState]
                                        toState:waitingReplyState];

    TKEvent *receiveConfirmEvent = [TKEvent
eventWithName:CUGestGameManagerReceiveConfirmEvent
                        transitioningFromStates:@[waitingReplyState]
                                        toState:goState];
```

```
    TKEvent *disconnectedEvent = [TKEvent
eventWithName:CUGestGameManagerDisconnectedEvent
                                transitioningFromStates:nil
                                                toState:idleState];
```

定义过程（**Guest**）

```
- (void)joinGame {

    if (![self.stateMachine.currentState.name isEqual:@"idle"]) {
      [self fireEvent:CUGestGameManagerDisconnectedEvent
userInfo:nil];
    }

    [self fireEvent:CUGestGameManagerSearchingEvent userInfo:nil];
}

- (void)searchingGames {
    AVQuery *query = [AVQuery
queryWithClassName:@"waiting_join_Ids"];
    [query orderByDescending:@"updatedAt"];
    [query setLimit:1];

    [query findObjectsInBackgroundWithBlock:^(NSArray *objects, NSEr-
ror *error) {
```

```
        NSMutableArray *installationIds = [[NSMutableArray alloc] init];
        for (AVObject *object in objects) {
          if ([object objectForKey:@"peerId"]) {
            [installationIds addObject:[object objectForKey:@"peerId"]];
          }
        }

        [self.session watchPeerIds:installationIds];

        [NSObject cancelPreviousPerformRequestsWithTarget:self
selector:@selector(sendJoinRequest) object:nil];
        [self performSelector:@selector(sendJoinRequest)
                   withObject:nil
                   afterDelay:2.0f];
      }];
    }

    - (void)sendJoinRequest {

      for (NSString *item in self.session.watchedPeerIds) {
        AVMessage *message = [AVMessage
messageForPeerWithSession:self.session
                                                  toPeerId:item
                                                   payload:@"join"];
        [self.session sendMessage:message transient:YES];
```

```
    }
}

- (void)sendGo{
    AVMessage *message = [AVMessage
messageForPeerWithSession:self.session
                                                toPeerId:self.otherPeerId
                                                 payload:@"go"];
    [self.session sendMessage:message transient:YES];
}
```

最后

state machine 是一个蛮厉害的锤子，只要是一个工具，就肯定会被滥用。。。state machine最大的好处是在于，方便我们思考清楚所有细节，主线，和错误流程。避免因为考虑不周全而产生的bug。结合之前的information flow的思路，会让我们的软件设计更加清楚。

demo code:https://github.com/studentdeng/ActivityGame

原文链接：http://studentdeng.github.io/blog/2014/11/05/ios-architecture2/

应用层的容错与分层设计

作者：Tim

针对在项目中碰到的一些容错设计问题，团队最近进行了一次技术沙龙，讨论了以下话题。

为什么需要应用层的容错设计?

一个完整的系统在内部是由很多小服务构成，服务之间以及服务与资源之间会存在远程调用。

- 每个系统的可用性不可能达到100%
- 各种网络及硬件问题，如网络拥堵、网络中断、硬件故障……
- 远程服务平均响应速度变慢

服务器平均响应速度如果慢下来，慢慢消耗掉系统所有资源，进而导致整个系统不可用。因此在分布式系统中，除了远程服务本身需要有容错设计之外，在应用层的远程调用的环节，需要有良好的容错设计。

应用层的容错设计有哪些方法？以下是微博团队使用过的一些实践。

访问MySQL的容错设计

- 写操作：如果master异常，直接抛异常。
- 读操作：如果slave有多个，先选择其中一个slave，如果获取连接失败，再选择其他的slave，如果全部不可用，最后选择master。

访问Memcached/Redis的容错设计

首先设置so_timeout，避免无限制等待；服务器连接如果IO异常，设置错误标志，一段时间停止访问；出错后定期主动（比如ping Redis）或被动（当被再次访问时）探测服务是否恢复。

Failover机制：
如果连接某个node失败, 当前pool启用一致性hash切换到backup node；如果backup node没有数据，则通过另外一个服务池（数据副本）获取数据。

访问远程HTTP API的容错设计

设置so_timeout；部分场景：短超时，重试一次；另外由于HTTP service情况的多样性，业务层面还有通用的降级机制。

访问不同资源使用不同方法存在的问题

从上面列举的部分场景来看，在访问不同资源时候，每种client访问都有一些相通的原理，但却要使用不同的重复实现。由于各个client独立实 现，实现时候由于各个远程服务协议及行为的差异，导致这些容错原理无法直接复用。另外在代码层面，不同的client也使用了不同年代的一些底层库，一些 早期client 的实现，数据层，连接层，协议层全部耦合在一起，也造成维护成本进一步加大。

比如之前一些服务开发中碰到的类似如下的问题：

- hbase-client 由于没有实现容错设计，导致访问出现了抖动，影响了同一服务池的其他调用，需要增加类似MySQL client的容错及快速失败策略；
- MySQL slave流量出现不均衡了，由于多个slave IP之间没有使用公用的负载均衡策略，因此需要重新添加、上线及验证。

另外目前分布式系统中大部分远程资源都是IO bound而不是CPU bound，而client大部分又是同步调用，造成大部分调用都在等待远程返回，同时也消耗了工作线程资源，以及大量线程context switch。

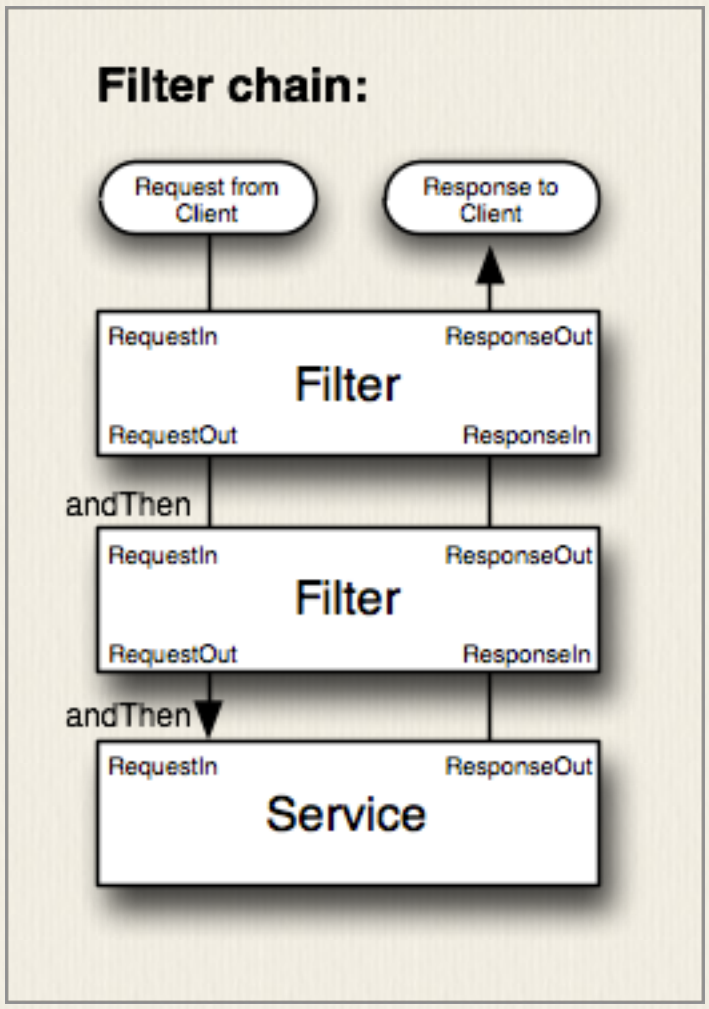
有没有可能统一的client?

这些策略原理上是可以公用的，能否出一个统一的client层来一劳永逸？不过这个需求不是twitter干过吗？

Finagle，不仅是平时理解的RPC框架，还有目标是想成为一个commons client，从另外一个层面，广义上访问远程资源也都可以理解成RPC，所以Finagle也常称为RPC框架。

Finagle implements uniform client and server APIs for several protocols, and is designed for high performance and concurrency.

在Twitter体系，分布式服务可以从future, service, filter三个层次理解，容错、超时、授权、tracing、重试等机制都是体现在filter中；而future则将client从多线程、队列、连 接池、资源管理释放出来，从关注控制流到关注数据流。并且默认变成异步方式。



Finagle的FailFast模块会避免分发请求到出现问题的服务，它通过来记录到每个host的错误来进行标记，当出错以后，Finagle会通过一个后台线程定期重连以检查是否恢复。当host宕机时，相关的service会标记成不可用。

如果来redisign一个通用的网络client，它应该包括哪些元素?

- 具有服务的分层设计，借鉴Future/Service/Filter概念
- 具有网络的分层设计，区分协议层、数据层、传输层、连接层
- 独立的可适配的codec层，可以灵活增加HTTP，Memcache，Redis，MySQL/JDBC，Thrift等协议的支持。
- 将多年各种远程调用High availability的经验融入在实现中，如负载均衡，failover，多副本策略，开关降级等。
- 通用的远程调用实现，采用async方式来减少业务服务的开销，并通过future分离远程调用与数据流程的关注。
- 具有状态查看及统计功能
- 当然，最终要的是，具备以下通用的远程容错处理能力，超时、重试、负载均衡、failover……

原文链接：http://timyang.net/service/application-failure-managment/

天猫11.11：搜索引擎实时秒级更新

作者：郭蕾

搜索是很多用户在天猫购物时的第一入口，搜索结果会根据销量、库存、人气对商品进行排序，而商品的显示顺序往往会决定用户的选择，所以保证搜索结果 的实时性和准确性非常重要。在电商系统中，特别是在“双十一”这样的高并发场景下，如何准确展示搜索结果显得尤为重要。在今年的“双十一”活动 中，InfoQ有幸采访到了阿里巴巴集团搜索引擎的三位负责人仁基、桂南和悾傅，与他们共同探讨了搜索引擎背后的细节。以下内容根据本次采访整理而成。

阿里巴巴的搜索引擎承担着全集团的搜索业务，包括淘宝、天猫、1688等系统，对比传统的搜索引擎，阿里集团的搜索引擎有一些比较大的突破性、创造 性的工作。传统的搜索引擎，只可以做到离线全量、增量构建索引，而阿里的搜索引擎已经是演变成为一个能够做到离线、增量、实时三个等级的搜索引擎。电商平 台最大的一个特点就是短时高并发，像双十一这样的活动中，搜索引擎需要考虑如何让流量发挥更大的价值。传统的搜索引擎解决短时高并发的思路是添加缓存层以 减少搜索引擎的访问量，而这样的解决方案，天猫之前也有使用，但是缓存会有延迟，实时搜索的需求根本无法满足。所以为了解决实时的问题，阿里的搜索引擎去 掉了应用层和业务层的缓存，重点优化和提升引擎层的服务能力。为了兼顾实时性和吞吐量，搜索引擎实现了全量、增量、实时三种更新通路。通过三种方式的灵活 组合，在保证了海量数据定期全量更新的同时提供了秒级实时更新能力，避免了数据延迟，提升了用户体验。

从整体上来看，阿里搜索引擎的架构图如下。从上到下，分别是应用层、业务层、搜索引擎层、离线处理层和DB层，应用层其实就是调用方，大的来看可以 分为Web、App、Wap。业务层会针对相应的业务对搜索结果进行整理，如Android和iOS的搜索结果显示是不一样的。搜索引擎层有

点类似传统系 统的搜索引擎，阿里巴巴的搜索引擎会在搜索的基础上根据用户习惯提供个性化的搜索结果。索引层主要包括全量索引和流式计算，全量索引其实就是一个基于 Hadoop/HBase的离线集群，而流式计算是阿里自己研发的一套系统。之所以没有选用Storm，是因为在这一层中，光有计算是不够的，还需要有数 据的存储（开源解决方案HBase）。如果使用Storm，接下来会面临一个问题，Storm是一个集群，HBase又会是一个集群，这样，Storm的 Disk以及HBase 的CPU其实都没有充分利用到，所以阿里的方案是Hadoop Yarn与HBase混合部署，把两个集群合并在一起，既可以做大规模的数据处理，也可以做流式计算，通过这样的方式，可以将离线和实时计算更好地融合。 最底层的数据源层会把用户、商品、交易信息同步到上层的HBase集群中。

Storm是一个无状态的流式计算框架，而无状态的流式计算体系，更适合做简单的统计分析，比如针对成交维度或者点击维度做计数。而阿里自研的流式 计算框架iStream，已经不再是简单的、无状态的流式计算概念。iStream借助HBase集群存储用户状态，以完成一些相对复杂的模型的计

算。同 时模型的计算结果可以通过相应的接口直接推送到上层的搜索引擎中，以服务每一条流量的排序变化。

在搜索引擎层，不仅包括商品的搜索引擎，还会包括其它层面的服务（如架构图所示）。商品搜索引擎中包含商品、店铺、活动等维度的信息，而图中的个性 化服务旨在为用户提供个性化的搜索体验，个性化服务会根据用户的实时行为反馈搜索结果。而QP（Query Planner）会对用户的搜索请求进行分析（搜索词、搜索场景、页面）进一步个性化搜索服务。在搜索引擎层，通过这三个系统的互相配合为上层的业务层提 供个性化的搜索数据。

不同的业务对应的搜索排序结果不同，阿里搜索引擎中排序部分是通过类似链式处理的方式实现的，内部称为排序链。排序链是由不同的用户特征对应的算法 插件组合而成，算法插件是单独存在的，可以根据具体情况组合到不同业务的排序链中。目前在线上运行的排序链有几十条，系统会根据不同的业务、用户、场景、 Query选择不同的排序逻辑

而在双十一这样的高并发活动中，搜索引擎需要保证流量的合理分配，比如搜索结果中不能显示售罄的商品。但是对于一些热门商品，从库存充足到售罄可能 是几分钟的时间。为了保证搜索结果的实时性，阿里搜索引擎架构针对这样的场景做了优化，去掉了不能感知业务变化的缓存（业务层），重点优化搜索引擎层的缓 存。以商品售罄的场景为例，当商品售罄时，业务系统会发送异步消息通知离线集群，离线集群通过流式计算将更新同步到引擎，而当引擎返回搜索结果时，会在缓 存的基础上对结果进行二次过滤，从而保证搜索结果的实时性和准确性。

另外，在今年双十一中，天猫搜索底层第一次使用精确到更新粒度的SKU（Stock Keeping Unit）引擎代替之前的宝贝引擎，底层引擎索引量较之前翻了几番。天猫从召回逻辑、前端的属性展示、筛选以及搜索结果页到详情页的联动，向用户提供了精 确度更高、更细致的搜索购物体验。对于标类产品，基于SKU引擎的搜索导购缩短了用户的搜索购物路径，比如搜索iPhone 5s后，SKU引擎会显示对应的销售属性，方便用户选择。此外在SKU引擎的基础上，天猫还实现了用户的尺码个性化，在包含确定尺码信息的类目中，如鞋、 文胸，天猫可以匹配用户的尺码个性化信息，将适合的商品优先展示给用户。

原文链接：http://www.infoq.com/cn/news/2014/11/tmall-1111-search-engine